IMPLEMENTASI *MULTIPATH ROUTING* BERBASIS ALGORITME DFS YANG DIMODIFIKASI

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Uis Yudha Tri Wirawan
NIM: 145150200111176

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2018

PENGESAHAN

IMPLEMENTASI *MULTIPATH ROUTING* BERBASIS ALGORITME DFS YANG DIMODIFIKASI

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Uis Yudha Tri Wirawan
NIM: 145150200111176

Skripsi ini telah diuji dan dinyatakan lulus pada
4 Mei 2018
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Widhi Yahya, S.Kom., M.Sc.	Achmad Basuki, S.T, M.Mg, Ph.D.
NIK: 2016078911211001	NIP: 19741118 200312 1 002

Mengetahui
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T, M.T, Ph.D.
NIP: 19710518 200312 1 001

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 22 Mei 2018

Uis Yudha Tri Wirawan

NIM: 145150200111176

KATA PENGANTAR

Puji syukur kehadirat Allah SWT yang telah memberikan rahmad, taufik dan hidayah-Nya sehingga laporan skripsi yang berjudul "Implementasi *Multipath Routing* Berbasis Algoritme DFS yang dimodifikasi" ini dapat terselesaikan

Penulis menyadari bahwa skripsi ini tidak akan berhasil tanpa bantuan dari beberapa pihak. Oleh karena itu, penulis ingin menyampaikan rasa hormat dan terima kasih kepada:

1. Bapak Widhi Yahya, S.Kom., M.Sc. dan Bapak Achmad Basuki, S.T, M.Mg, Ph.D. selaku dosen pembimbing skripsi yang telah dengan sabar membimbing dan mengarahkan penulis sehingga dapat menyelesaikan skripsi ini.
2. Bapak Tri Astoto Kurniawan, S.T, M.T, Ph.D. selaku Ketua Jurusan Teknik Informatika Fakultas Ilmu Komputer Universitas Brawijaya.
3. Bapak Agus Wahyu Widodo, S.T, M.Cs. selaku Ketua Program Studi Teknik Informatika
4. Kedua orang tua dan seluruh keluarga besar atas segala nasehat, kasih sayang, perhatian dan kesabarannya di dalam membesarkan dan mendidik penulis, serta yang senantiasa tiada henti-hentinya memberikan doa dan semangat demi terselesaikannya skripsi ini.
5. Teman-teman satu angkatan Program Studi Informatika 2014 tercinta yang selalu memberikan informasi, semangat, dorongan dan bantuan pikiran.
6. Seluruh civitas akademika Fakultas Ilmu Komputer Universitas Brawijaya yang telah banyak memberi bantuan dan dukungan selama penyelesaian skripsi ini.
7. Semua pihak yang telah membantu dan berbagi ilmu dalam penyelesaian skripsi ini yang tidak dapat penulis sebutkan satu per satu.

Penulis menyadari bahwa dalam penyusunan skripsi ini masih banyak kekurangan, sehingga saran dan kritik yang membangun sangat penulis harapkan. Akhir kata penulis berharap skripsi ini dapat membawa manfaat bagi semua pihak yang menggunakannya.

Malang, 22 Mei 2018

Uis Yudha Tri Wirawan

uisyudha@student.ub.ac.id

ABSTRAK

Uis Yudha Tri Wirawan, Implementasi *Multipath Routing* Berbasis Algoritme DFS yang dimodifikasi

Pembimbing : Widhi Yahya, S.Kom., M.Sc. dan Achmad Basuki, S.T, M.Mg, Ph.D.

Multipath routing merupakan metode *routing* dengan menggunakan beberapa jalur yang tersedia. Dengan *multipath routing* diharapkan dapat mengurangi kemacetan pada jaringan dengan melakukan *load-balancing* sehingga *traffic* jaringan dapat didistribusikan ke beberapa jalur. *Multipath routing* dapat diterapkan dengan konsep *Software-Defined Networking* (SDN) yang memisahkan *control plane* dan *data plane* sehingga dapat mempermudah pengembangan aplikasi jaringan. Untuk itu, penulis melakukan implementasi *multipath routing* pada jaringan OpenFlow SDN berbasis algoritma DFS yang dimodifikasi dan algoritma Dijkstra yang dimodifikasi sebagai referensi pembanding. Hasil simulasi menunjukkan sistem dapat melakukan pencarian beberapa jalur pada suatu topologi dengan *response time* rata-rata algoritme DFS adalah 612,7 *ms.* Rata-rata *response time* algoritme DFS yang dimodifikasi adalah 632,6 *ms* dan Dijkstra 325,6 *ms*. Jumlah iterasi Algoritme DFS memerlukan 12694 iterasi, *modified*-DFS sebanyak 12860 iterasi, dan Dijkstra sebanyak 800 iterasi. Rata-rata *execution time* algoritme DFS adalah 0,0185 *ms, modified* DFS adalah 0,0258 *ms*, dan *modified* Dijkstra 0,0005 *ms.* Hasil pengujian *throughput* menunjukan sebanyak 50% dari *client* mendapat *throughput* 20 *Mbps* atau kurang pada algoritme *modified*-DFS. Sedangkan pada Dijkstra, sebanyak 50% dari *client* mendapat *throughput* 23 *Mbps* atau kurang. Hasil pengujian *multipath* menunjukkan mekanisme *load-balancing* dengan *group table* masih terdapat *flow* yang tidak terdistribusi seimbang.

Kata kunci: *multipath routing, software-defined networking*, OpenFlow.

ABSTRACT

Multipath routing is the routing technique that use some of the available paths. Multipath routing is expected to reduce congestion on the network with load-balancing so that network traffic can be distributed to several available paths. It can be implemented with the concept of Software-Defined-Networking (SDN) which enable decoupling of control plane and data plane so as to facilitate the development of network applications. Here, we perform multipath routing implementation on the OpenFlow SDN and based on modified DFS and modified Djikstra as reference. The simulation results show that the system can perform multiple path search on a topology with the average response time of DFS algorithm is 612,7 ms. The average response time of modified-DFS algorithm is 632.6 ms and Dijkstra 325.6 ms. Number of iterations of the DFS algorithm requires 12694 iterations, modified-DFS requires 12860 iterations, and Dijkstra requires 800 iterations. The average execution time of DFS algorithm 0,0185 *ms, modified* DFS is 0,0258 *ms*, *and modified* Dijkstra 0,0005 *ms. The result of throughput test shows that 50% of clients get throughput of 20 Mbps or less on modified-DFS algorithm. While at Dijkstra, as many as 50% of clients get throughput of 23 Mbps or less. The multipath test results show that load-balancing mechanism with group table still has unbalance distributed flow.*

Keywords: *multipath routing, software-defined networking*, OpenFlow.

DAFTAR ISI

PENGESAHAN ..... **ii**
PERNYATAAN ORISINALITAS ..... **iii**
KATA PENGANTAR ..... **iv**
ABSTRAK ..... **v**
ABSTRACT ..... **vi**
DAFTAR ISI ..... **vii**
DAFTAR TABEL ..... **ix**
DAFTAR GAMBAR ..... **x**
DAFTAR LAMPIRAN ..... **xii**
BAB 1 PENDAHULUAN **1**
1.1 Latar Belakang 1
1.2 Rumusan Masalah 2
1.3 Tujuan 2
1.4 Manfaat 2
1.5 Batasan Masalah 2
1.6 Sistematika Pembahasan 2
BAB 2 LANDASAN KEPUSTAKAAN **4**
2.1 Kajian Pustaka 4
2.2 Dasar Teori 4
2.2.1 *Software-Defined Networking* 4
2.2.2 *Multipath Routing* 7
2.2.3 Depth-First Search 7
2.2.4 Dijkstra 8
2.2.5 *Load Balancing* 8
2.2.6 Topologi *Fat Tree* 9
2.2.7 Ryu 9
2.2.8 Mininet 9
2.2.9 OpenvSwitch 9
2.2.10 sFlow-RT 10
BAB 3 METODOLOGI **11**
3.2 Studi Literatur 12

3.3 Analisis Kebutuhan 12
3.3.1 Kebutuhan Fungsional 12
3.3.2 Kebutuhan Non-fungsional 12
3.4 Perancangan Sistem 13
3.5 Implementasi 14
3.6 Pengujian dan Analisis 14
3.7 Pengambilan Kesimpulan dan Saran 14
BAB 4 PERANCANGAN DAN IMPLEMENTASI 15
4.1 Perancangan Sistem 15
4.1.1 Perancangan Topologi 16
4.1.2 Perancangan Algoritme Pencarian Jalur 17
4.1.3 Perancangan Metrik Penilaian Jalur 22
4.1.4 Perancangan Algoritme *Load-Balancing* 23
4.1.5 Perancangan Instalasi Jalur 23
4.2 Implementasi 23
4.2.1 Instalasi 23
4.2.2 Membangun Topologi di *Mininet* 25
4.2.3 Langkah - Langkah Menjalankan Sistem 27
4.2.4 Pengembangan Program *Controller* 28
BAB 5 PENGUJIAN DAN ANALISIS 35
5.1 Pengujian Pencarian Jalur 35
5.1.1 Pencarian Jalur Secara Manual 40
5.2 Pengujian *Response Time* 45
5.3 Pengujian Jumlah Iterasi 46
5.4 Pengujian *Execution Time* 47
5.5 Pengujian *Throughput* 47
5.6 Pengujian *Multipath* 48
BAB 6 Penutup 50
6.1 Kesimpulan 50
6.2 Saran 50
DAFTAR PUSTAKA 51
LAMPIRAN 53

DAFTAR TABEL

Tabel 2.1 Kajian Pustaka 4
Tabel 4.1 Daftar Komponen Miniedit 25
Tabel 4.3 Variabel dan Struktur Data 28
Tabel 4.4 Variabel dan Struktur Data (Lanjutan) 29
Tabel 4.5 Kode Sumber Algoritme Pencarian Jalur 29
Tabel 4.6 Kode Sumber Algoritme Pencarian Jalur (Lanjutan) 30
Tabel 4.7 Kode Sumber Algoritme Pencarian Jalur (Lanjutan) 31
Tabel 4.8 Kode Sumber Algoritme Penilaian Jalur 31
Tabel 4.9 Kode Sumber Algoritme Pemilihan Jalur 32
Tabel 4.10 Kode Sumber *Monitoring Traffic* 32
Tabel 4.11 Kode Sumber Instalasi Jalur 33
Tabel 4.12 Kode Sumber Instalasi Jalur (Lanjutan) 34
Tabel 5.1 Hasil Pengujian Pencarian Jalur 40
Tabel 5.2 Proses Seleksi Jalur *Modified* DFS 43
Tabel 5.4 Hasil Pengujian *Response Time* 45
Tabel 5.5 Hasil Pengujian Jumlah Iterasi 46
Tabel 5.6 Hasil Pengujian *Execution Time* 47
Tabel 5.7 Hasil Pengujian *Multipath* 49

DAFTAR GAMBAR

Gambar 2.1 Arsitektur *Software-Defined Networking* 5
Gambar 2.2 Kompenen Utama OpenFlow *Switch* 6
Gambar 2.3 Algoritme DFS..... 7
Gambar 2.4 Algoritme Dijkstra 8
Gambar 2.5 Topologi *Fat Tree*..... 9
Gambar 2.6 sFlow-RT 10
Gambar 3.1 Diagram Alir Metode Penelitian..... 11
Gambar 3.2 Tahap Perancangan Sistem 13
Gambar 4.1 Blok Diagram Sistem 15
Gambar 4.2 Contoh Topologi Jaringan 16
Gambar 4.3 Topologi *Fat Tree*..... 17
Gambar 4.4 Topologi *Fat-Tree* pada Mininet..... 17
Gambar 4.5 Diagram Alir Modifikasi DFS..... 18
Gambar 4.6 Diagram Alir Modifikasi DFS (Lanjutan) 19
Gambar 4.7 Modifikasi Algoritme DFS..... 20
Gambar 4.8 Diagram Alir Modifikasi Algoritme Dijkstra..... 21
Gambar 4.9 Modifikasi Algoritme Dijkstra..... 21
Gambar 4.10 Algoritme Instalasi Jalur 23
Gambar 4.12 Contoh Pembangunan Topologi di Miniedit..... 26
Gambar 4.13 Pengaturan *Link Bandwidth* 26
Gambar 4.14 Pengaturan *Controller Pada Miniedit* 27
Gambar 4.15 Pengaturan *Preferences* Pada *Miniedit*..... 27
Gambar 5.1 Hasil Pencarian Jalur *Single*-path Menggunakan DFS 35
Gambar 5.2 Hasil Pencarian Jalur *Single*-path Menggunakan *Modified* DFS..... 35
Gambar 5.3 Hasil Pencarian Jalur *Single*-path Menggunakan Dijkstra..... 36
Gambar 5.4 Hasil Pencarian Jalur *Multipath* Menggunakan DFS 36
Gambar 5.5 Hasil Pencarian Jalur *Multipath* Menggunakan *Modified* DFS..... 36
Gambar 5.6 Hasil Pencarian Jalur *Multipath* Menggunakan Dijkstra 37
Gambar 5.7 Hasil Instalasi *Flow* 37
Gambar 5.8 Hasil Instalasi *Group Table* 37
Gambar 5.9 Hasil Pencarian Jalur *Single-path* Menggunakan DFS..... 38

Gambar 5.10 Hasil Pencarian Jalur *Single-path* Menggunakan *Modified* DFS 38
Gambar 5.11 Hasil Pencarian Jalur *Single-path* Menggunakan Dijkstra..... 38
Gambar 5.12 Hasil Pencarian Jalur *Multipath* Menggunakan DFS 39
Gambar 5.13 Hasil Pencarian Jalur *Multipath* Menggunakan *Modified* DFS..... 39
Gambar 5.14 Hasil Pencarian Jalur *Multipath* Menggunakan Dijkstra 39
Gambar 5.15 Topologi *Fat Tree* 2 *Level* 40
Gambar 5.16 Proses Pencarian Jalur *Modified* DFS 41
Gambar 5.17 Proses Pencarian Jalur *Modified* DFS (Lanjutan)..... 42
Gambar 5.18 Proses Pencarian Jalur Dijkstra Iterasi 1 44
Gambar 5.19 Proses Menghapus Jalur dari Graf Dijkstra Iterasi 1 44
Gambar 5.20 Proses Pencarian Jalur Dijkstra Iterasi 2 44
Gambar 5.21 Proses Menghapus Jalur dari Graf Dijkstra Iterasi 2 45
Gambar 5.22 Contoh Hasil Ping dari h1 ke h2 45
Gambar 5.23 Jumlah Iterasi Algoritme DFS 46
Gambar 5.24 Jumlah Iterasi Algoritme *Modified*-DFS 46
Gambar 5.25 Jumlah Iterasi Algoritme Dijkstra 46
Gambar 5.26 Contoh Hasil Waktu Eksekusi Pencarian Jalur dengan DFS 47
Gambar 5.27 Contoh Pengujian *Throughput* dengan 10 *Client*..... 48
Gambar 5.28 Hasil Pengujian *Throughput* dengan 40 *Client*..... 48
Gambar 5.29 Contoh Penggunaan tcpdump 49

DAFTAR LAMPIRAN

LAMPIRAN 53

A. Kode Program Ryu 53

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Multipath routing merupakan metode *routing* dengan menggunakan beberapa jalur yang tersedia. Berbeda dengan *single-path routing* yang hanya menggunakan satu jalur dari sumber ke tujuan. Pada algortime *single-path routing*, keseluruhan jalur yang ada pada topologi jaringan yang disebut dengan kemampuan *multipath* tidak dapat sepenuhnya di manfaatkan. Seperti pada algoritme *spanning tree,* topologi jaringan selalu dipotong dan dikurangi menjadi bentuk *tree* sehingga kemampuan *multipath* pada topologi yang memiliki beberapa cabang redundan tidak dapat dimanfaatkan dan menyebabkan pemborosan sumber daya jaringan (Lei, et al., 2015). Dengan *multipath routing* topologi jaringan yang memiliki cabang redundan dapat dimanfaatkan sepenuhnya. Selain itu, *multipath routing* dapat mengurangi kemacetan (*congestion)* pada jaringan dengan melakukan *load-balancing* sehingga *traffic* jaringan dapat di distribusikan dengan memilih jalur yang menyediakan sumber daya paling besar (Joe, et al., 2014).

Implementasi *multipath* routing pada arsitektur jaringan saat ini sangat sulit karena tidak ada antarmuka untuk melakukan eksperimen dan pengembangan. Solusi dari permasalahan tersebut dapat diselesaikan dengan menggunakan konsep jaringan atau paradigma yang disebut *Software-Defined Networking* (SDN). SDN adalah konsep atau paradigma jaringan yang memiliki sebuah perangkat lunak utama yang disebut *controller* sebagai penentu perilaku keseluruhan jaringan. Pada SDN, *control plane* yaitu kecerdasan yang diterapkan di *controller* dipisahkan dengan *data plane* yaitu perangkat yang melakukan penerusan paket (Kim & Feamster, 2013). Berbeda dengan arsitektur jaringan saat ini, *control plane* dan *data plane* masih terikat dengan vendor masing-masing perangkat sehingga tidak mudah untuk melakukan pengembangan jaringan. Pada arsitektur jaringan saat ini masih bersifat terdistribusi, hal ini mengakibatkan pengelolaan dan konfigurasi jaringan dilakukan di tiap perangkat (Kreutz, et al., 2015). Salah satu perwujudan SDN yang sudah banyak digunakan saat ini yaitu OpenFLow.

Pada penelitian sebelumnya yaitu berjudul "*Multipath Routing* dengan *Load-Balancing* pada *OpenFlow Software-Defined Network*" telah menerapkan *multipath routing* yang memiliki kemampuan *load-balancing* dengan menggunakan fitur *group action* pada *OpenvSwitch* (Maulana, 2017). Penelitian tersebut menggunakan algoritme DFS yang dimodifikasi sehingga dapat menemukan seluruh jalur yang dapat digunakan dalam *multipath routing.* Namun pada penelitian tersebut menggunakan algoritme *naive* DFS sehingga masih terdapat jalur yang tidak independen antara jalur satu dengan yang lain masih ada kesamaan. Penelitian lain yang berjudul "*A Multipath Transmission Scheme for the Improvement of Throughput over SDN*" telah mengimplementasikan *multipath routing* pada SDN dengan menggunakan algoritme Dijkstra untuk pencarian

jalurnya. Proses pencarian jalur dengan cara melakukan iterasi sampai tidak ada lagi jalur yang ditemukan. Ketika sebuah jalur telah ditemukan, *link* dan *node* dihapus dari pencarian kecuali *node* sumber dan tujuan (Chiang, et al., 2017).

Dalam penelitian ini, penulis melakukan implementasi *multipath routing* berbasis algoritme DFS yang dimodifikasi pada emulasi jaringan SDN menggunakan Mininet dan Ryu *controller.* Parameter pengujian yang digunakan yaitu *response time*, jumlah iterasi, *execution time, throughput,* dan pengujian *multipath*. Selain itu digunakan algoritme Dijkstra yang dimodifikasi pada penelitian (Ciang, et al., 2017) sebagai referensi pembanding.

1.2 Rumusan Masalah

Berdasarkan latar belakang yang sudah dijabarkan, dirumuskan permasalahan sebagai berikut

1. Bagaimana menemukan n-jalur *multipath routing* pada jaringan OpenFlow menggunakan algoritme DFS yang dimodifikasi?
2. Bagaimana mendistribusikan beban jaringan dari jalur-jalur yang telah ditemukan pada jaringan OpenFlow?
3. Bagaimana kinerja dari *multipath routing* berbasis algoritme DFS yang dimodifkasi pada jaringan OpenFlow?

1.3 Tujuan

1. Mengimplementasikan algoritme DFS yang dimodifikasi sehingga dapat menemukan n-jalur dalam *mulipath routing* pada jaringan OpenFlow.
2. Mendistribuskan beban jaringan berdasarkan jalur-jalur yang tersedia.
3. Menganalisis kinerja dari *multipath routing* berbasis algoritme DFS yang dimodifikasi pada jaringan OpenFlow.

1.4 Manfaat

1. Dapat meningkatkan pengetahuan teoritis dan aplikatif bagi penulis.
2. Dapat digunakan sebagai referensi untuk penelitian lain.
3. Dapat meningkatkan kinerja dari jaringan.

1.5 Batasan Masalah

Agar permasalahan dapat terarah dengan baik, maka dalam penelitian ini dibatasi dalam hal:

1. Implementasi SDN yang digunakan adalah OpenFlow.
2. Menggunakan *controller* Ryu sebagai *controller* SDN.
3. Dilakukan pada emulasi jaringan Mininet.

1.6 Sistematika Pembahasan

Penulisan sistematika pembahasan pada masing-masing bab dapat dijelaskan sebagai berikut.

BAB 1 PENDAHULUAN

Pada bab pendahuluan menjelaskan mengenai latar belakang, rumusan masalah, tujuan, manfaat, batasan masalah, dan sistematika pembahasan penelitian ini tentang implementasi *multipath routing* berbasis DFS dan Dijkstra pada jaringan OpenFlow.

BAB 2 LANDASAN KEPUSTAKAAN

Pada bab ini terdapat beberapa kajian terkait penelitian-penelitian yang sudah dilakukan sebelumnya serta pemaparan teori dan konsep sebagai dasar dari penelitian ini.

BAB 3 METODOLOGI

Pada bab ini dijelaskan mengenai metodologi yang digunakan dalam penelitian yaitu studi pustaka, analisis kebutuhan, perancangan sistem, implementasi dan pengujian.

BAB 4 PERANCANGAN DAN IMPLEMENTASI

Pada bab ini membahas mengenai perancangan dan implementasi *multipath routing* berbasis DFS dan Dijkstra pada jaringan OpenFlow

BAB 5 PENGUJIAN DAN ANALISIS

Pada bab ini menjelaskan mengenai langkah-langkah pengujian, hasil pengujian, dan pembahasan kinerja *multipath routing* berbasis DFS dan Dijkstra.

BAB 6 PENUTUP

Pada bab ini berisi kesimpulan dari hasil penelitian yang dibuat berdasarkan hasil pengujian dan pembahasan terhadap sistem yang dibangun serta beberapa saran yang dapat digunakan dalam pengembangan berikutnya.

BAB 2 LANDASAN KEPUSTAKAAN

2.1 Kajian Pustaka

Tabel 2.1 Kajian Pustaka

No	Nama Penulis, Tahun, dan Judul	Persamaan	Perbedaan	
			Penelitian Terdahulu	Rencana Penelitian
1	Yi-Rou Chiang; Chih-Heng Ke; Yun-Shuai Yu, 2017, *A Multipath Transmission Scheme for the Improvement of Throughput over SDN*	Mengimplementasikan *multipath routing* dengan *load balancing* pada jaringan OpenFlow dengan algoritme Dijkstra	Parameter pengujian yang digunakan hanya *throughput*	Parameter pengujian yang dilakukan yaitu pencarian jalur, *response time,* jumlah iterasi, *execution time*, *throughput,* dan pengujian *multipath*
2	Wildan Maulana S., 2017, *Multipath Routing* dengan *Load-Balancing* Pada *OpenFlow Software-Defined Network*	Mengimplementasikan *multipath routing* dengan *load balancing* pada jaringan OpenFlow dengan algoritme DFS	Pencarian rute menggunakan algoritme *naive* DFS, sehingga masih terdapat jalur yang memiliki persamaan	Memodifikasi algoritme DFS dengan mengurutkan berdasarkan jalur terpendek dan tidak ada jalur yang sama

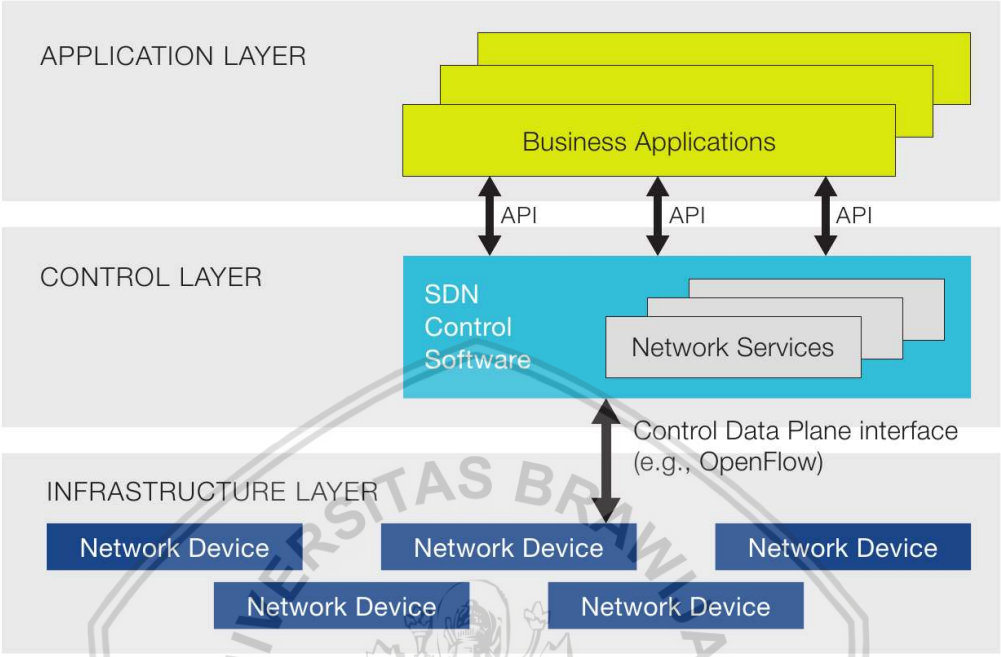
2.2 Dasar Teori

Pada sub bab ini akan dijelaskan beberapa teori sebagai pendukung penelitian ini, yaitu beberapa teori dalam mengembangkan *multipath routing* berbasis DFS dan Dijkstra pada jaringan OpenFlow.

2.2.1 *Software-Defined Networking*

Software-Defined Networking (SDN) adalah paradigma jaringan baru dengan memisahkan *control plane* sebagai kecerdasan dari jaringan yang mengatur dan menentukan bagaimana paket dikirim dan *data plane* sebagai perangkat yang bertugas menjalankan aksi yang ditentukan oleh *control plane*. Pada SDN, suatu jaringan dapat dipandang secara tersentralisasi sehingga memudahkan dalam

melakukan manajemen. Dengan pemisahan *control plane* dan *data plane* perilaku dari suatu jaringan dapat diprogram melalui antarmuka terbuka sehingga dapat dilakukan inovasi dan pengembangan pada jaringan tersebut (Nunes, et al., 2014).



Gambar 2.1 Arsitektur *Software-Defined Networking*

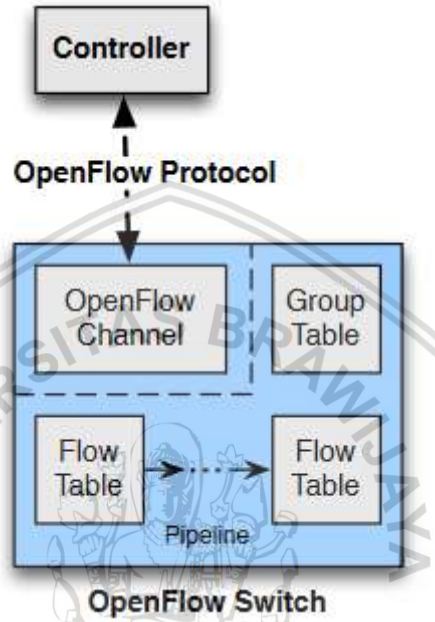
Sumber: opennetworking.org

Menurut (Kreutz, et al., 2015) , arsitektur jaringan SDN di definisikan ke dalam empat pilar.

1. Dipisahkannya *control plane* dan *data plane*. Pada perangkat jaringan fungsionalitas kontrol dihapus dan hanya menjadi perangkat sederhana yang bertugas meneruskan paket (*forwarding)*
2. Pengambilan keputusan penerusan bukan berdasarkan tujuan paket tetapi berdasarkan *flow*. Sebuah *flow* secara luas dapat didefinisikan sebagai sekumpulan kriteria atau *filter* dari paket dan sekumpulan aksi atau instruksi.
3. Logika kontrol dipindahkan pada entitas terpisah yang disebut sebagai NOS atau SDN *controller.* NOS (*Network Operating System)* adalah platform perangkat lunak yang berjalan pada teknologi server yang menyediakan sumber daya dan *library* untuk memfasilitasi pemrograman dari perangkat *forwarding* berdasarkan tampilan abstraksi jaringan yang tersentralisasi. Tujuan dari NOS serupa dengan sistem operasi tradisional.
4. Jaringan dapat diprogram menggunakan aplikasi perangkat lunak yang berjalan di atas NOS yang berinteraksi dengan perangkat *forwarding (switch)* yang dicakupinya.

2.2.1.2 OpenFlow

OpenFlow adalah standar protokol atau antarmuka komunikasi yang didefinisikan di antara lapisan kontrol (*control plane*) dan *forwading* (*data plane*) pada arsitektur SDN. OpenFlow memungkinkan akses langsung dan mengubah *forwading plane* dari perangkat jaringan seperti *switch* dan *router*, baik fisik maupun virtual. OpenFlow menyediakan antarmuka terbuka untuk memprogram *data plane* (Open Networking Fundation, 2012).



Gambar 2.2 Kompenen Utama OpenFlow *Switch*

Sumber: opennetworking.org

Pada sebuah OpenFlow *switch* terdiri satu atau lebih *flow table* dan *group table* yang melakukan pencarian dan penerusan paket, dan *OpenFlow channel* ke *controller*. *Switch* berkomunikasi dengan *controller* dan *controller* mengelola *switch* melalui protokol OpenFlow.

Dengan menggunakan protokol OpenFlow, *controller* dapat menambahkan, memperbarui, dan menghapus *flow entries* pada *flow table,* secara reaktif dan proaktif. Setiap *flow table* pada *switch* berisi sekumpulan *flow entries*, setiap *flow entry* terdiri dari *match field, counters,* dan kumpulan instruksi untuk diterapkan pada paket yang sesuai.

Pada *group tables* berisi sekumpulan *group entries* yang terdiri dari *group id, group type, counters,* dan *action buckets.* Beberapa tipe yang tersedia pada *group table* adalah:

a. *all,* digunakan untuk *multicast* dan *broadcast*
b. *select,* digunakan untuk *multipath* dan *load-balancing*
c. *indirect,* tipe ini hanya menjalankan satu *bucket.*
d. *fast failover,* tipe ini akan menjalankan *first live bucket.*

2.2.1.3 *Controller*

Sistem operasi tradisional menyediakan abstraksi (*high level* API) untuk mengakses perangkat-perangkat dengan level lebih bawah seperti *hard drive,* CPU, dan memori. Sedangkan pada perangkat jaringan saat ini masih dikelola dan dikonfigurasi sesuai dengan instruksi masing-masing vendor (Cisco, Juniper) dan kebanyakan tidak menyediakan antarmuka untuk melakukan pengembangan. *Controller* pada SDN adalah sebuah aplikasi yang merupakan otak dari jaringan dan memberikan pandangan tersentralisasi untuk memungkinkan jaringan cerdas. Controller SDN berfungsi sebagai sistem operasi jaringan untuk mengelola *flow control* pada *switch* melalui *shoutbound* API seperti OpenFlow. *Controller* SDN menyediakan *northbound* API yang digunakan untuk mengembangkan aplikasi jaringan (Kreutz, et al., 2015). Beberapa contoh *controller* SDN adalah Ryu, Floodlight, dan ONOS.

2.2.2 *Multipath Routing*

Multipath routing adalah alternatif dari *single-path routing* yang dapat mendistribusikan *traffic* jaringan melalui beberapa jalur yang terdapat pada jaringan (Lei, et al., 2015). Hal tersebut berbeda dengan metode *single-path routing* yang hanya melakukan penerusan paket menggunakan satu jalur saja. Dengan *multipath routing* dapat memberikan *throughput* yang lebih besar dan *congestion* dapat dihindari karena menggunakan lebih dari satu jalur antara sumber dan tujuan (Ramdhani, et al., 2016).

2.2.3 Depth-First Search

Depth-First Search (DFS) adalah algoritme *uninformed search* untuk penelusuran rute dengan cara melakukan ekspansi menuju *node* paling dalam pada sebuah graf. Setelah *node* selesai di ekspansi dilakukan *backtracking* untuk mengekspansi *node* yang lainnya sehingga dapat ditemukan seluruh jalur dari sumber ke tujuan. DFS dapat diimplementasikan menggunakan struktur data *stack*. Secara umum algoritme DFS dijabarkan seperti pada gambar 2.3 (Heap, 2002).

```
1   DFS( G, v ) ( v is the vertex where the search starts )
2       Stack S := {}; ( start with an empty stack )
3       for each vertex u, set visited[u] := false;
4       push S, v;
5       while (S is not empty) do
6          u := pop S;
7          if (not visited[u]) then
8             visited[u] := true;
9             for each unvisited neighbour w of u
10               push S, w;
11         end if
12      end while
13  END DFS()
```

Gambar 2.3 Algoritme DFS

2.2.4 Dijkstra

Dijkstra merupakan algoritme pencarian jalur terpendek dari sumber ke tujuan. Algoritme ini menggunakan strategi *greedy,* di setiap langkah dipilih sisi dengan bobot terkecil yang menghubungkan sebuah simpul yang sudah dipilih dengan simpul lain yang belum dipilih sehingga menghasilkan satu jalur terpendek. Secara umum algoritme Dijkstra dapat dijabarkan seperti pada kode sumber 2.2 (Heap, 2002).

```
1   DIJKSTRA( V, E, s )
2       for each v in V
3          d[v] := oo;   ("infinity")
4          pred[v] := NULL;
5       end for
6       d[s] := 0;
7       S := {};
8       V' := V;
9       while ( V' is not empty ) do
10         find a vertex u in V' such that d[u] is minimum;
11         V' := V' - {u};
12         S := S U {u};
13         for each edge e = (u,v) in E
14            if ( v is not in S ) and ( d[v] > d[u] + w(u,v) ) then
15               d[v] := d[u] + w(u,v);
16               pred[v] := u;
17            end if
18         end for
19      end while
20  END
```

Gambar 2.4 Algoritme Dijkstra

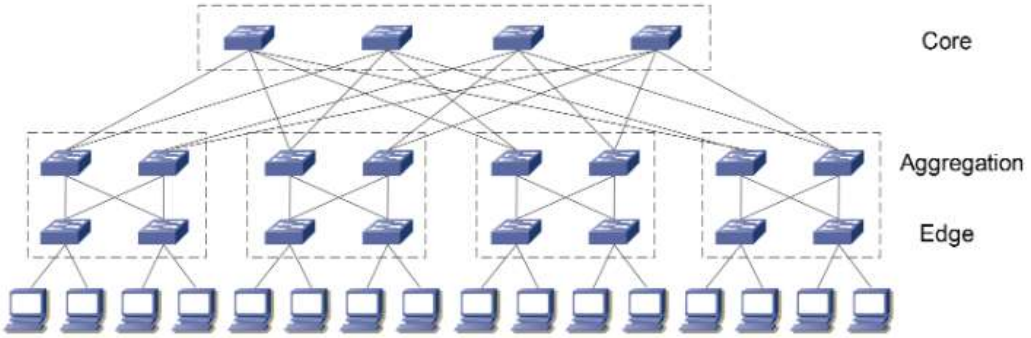
2.2.5 *Load Balancing*

Load balancing adalah metode dalam meningkatkan kinerja jaringan dengan membagi beban *traffic* dan mendistribusikan melalui beberapa jalur alternatif dengan setara (Joe, et al., 2014).

Beberapa algoritme *load balancing* yang sering digunakan adalah *round-robin*, *least-connection,* dan *least-loaded*. Algoritme penjadwalan *round-robin* bekerja dengan cara memberikan giliran pada *switch* atau *router* secara berurutan. Algoritme *least-connection* akan memilih *switch* atau *router* yang memiliki koneksi keluar (*outgoing traffic)* paling sedikit. Sedangkan pada algoritme *least-loaded* bekerja dengan memilih *switch* yang memiliki beban paling sedikit.

Penelitian yang dilakukan (Mustafa & Ibrahim, 2015) pada sebuah simulasi untuk membandingkan efisiensi algoritme *round-robin, least-connection,* dan *least-loaded* pada 8 HTTP server didapatkan bahwa *round-robin* dapat mendistribusikan beban dengan seimbang, kecuali pada salah satu server yang mendapatkan beban lebih banyak. *Least-connection* dapat mendistribusikan beban dengan variasi rendah. *Least-loaded* dapat mendistribusikan beban namun dengan variasi yang tinggi di antara 8 HTTP server tersebut. Selain itu, utilitas CPU dari server pada *least-loaded* lebih tinggi dari dua algoritme lainnya.

2.2.6 Topologi *Fat Tree*



Gambar 2.5 Topologi *Fat Tree*

Sumber: (Wang, et al., 2014)

Topologi *Fat Tree* diperkenalkan pertama kali oleh Al-Fares *et al* untuk membangun jaringan pusat data. Kemudian banyak penelitian yang menggunakan topologi ini dalam penelitian jaringan pusat data (Wang, et al., 2014). Topologi *Fat Tree* memiliki beberapa jalur dari satu *server* ke *server* lain, sehingga *multipath routing* dapat diterapkan pada topologi tersebut.

2.2.7 Ryu

Ryu merupakan kerangka kerja *Software Defined Networking* berbasis komponen. Ryu menyediakan komponen perangkat lunak sebagai API (*Aplication Programming Interface)* untuk memudahkan pengembang dalam membangun aplikasi-aplikasi kontrol dan manajemen jaringan (Ryu SDN Framework Community, 2017). Ryu mendukung OpenFlow protokol dari versi 1.0 hingga versi 1.5. Pengembangan aplikasi berbasis SDN menggunakan Ryu dapat dilakukan menggunakan bahasa pemrograman Python.

2.2.8 Mininet

Mininet adalah perangkat lunak emulator jaringan yang dapat digunakan untuk membangun dan mengemulasikan jaringan berupa *host* virtual, *switch, controller*, dan *link*. Minenet *host* dapat menjalankan program standar bawaan linux*, dan* mendukung OpenFlow (Mininet Team, 2017). Dengan Mininet emulasi jaringan dapat dilakukan pada sumber daya terbatas seperti sebuah laptop. Mininet dapat dijalankan menggunakan *CLI (Command Line Interface)* atau GUI (*Graphical User Interface)* dengan bantuan Miniedit. Untuk mengemulasikan jaringan SDN Mininet menggunakan virtual *switch* yaitu *OpenvSwitch.*

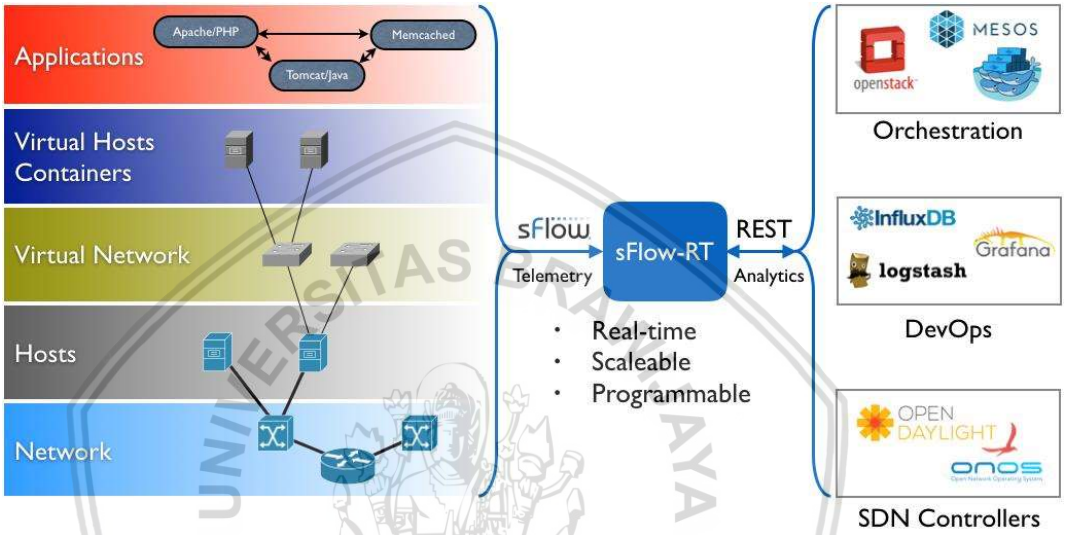
2.2.9 OpenvSwitch

OpenvSwitch merupakan *switch* virtual *multilayer* yang dilisensi di bawah lisensi *open source* Apache 2.0. *OpenvSwitch* mendukung beberapa antarmuka manajemen standar dan protokol seperti OpenFlow, NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1ag (Linux Fundation, 2016). Pada OpenvSwitch 2.5.2

mendukung OpenFlow hingga versi 2.5 yang menyediakan fitur *group action* untuk melakukan *load-balancing* di *switch.*

2.2.10 sFlow-RT

sFlow adalah alat yang dapat digunakan untuk *monitoring* jaringan secara *real-time* untuk mendukung pengembangan aplikasi SDN dengan menggunakan teknologi analitik asinkron. Hal tersebut dapat memungkinkan pengembangan aplikasi dengan memperhatikan kinerja misalnya *load-balancing* atau perlindungan terhadap serangan DDoS (InMon, 2017).



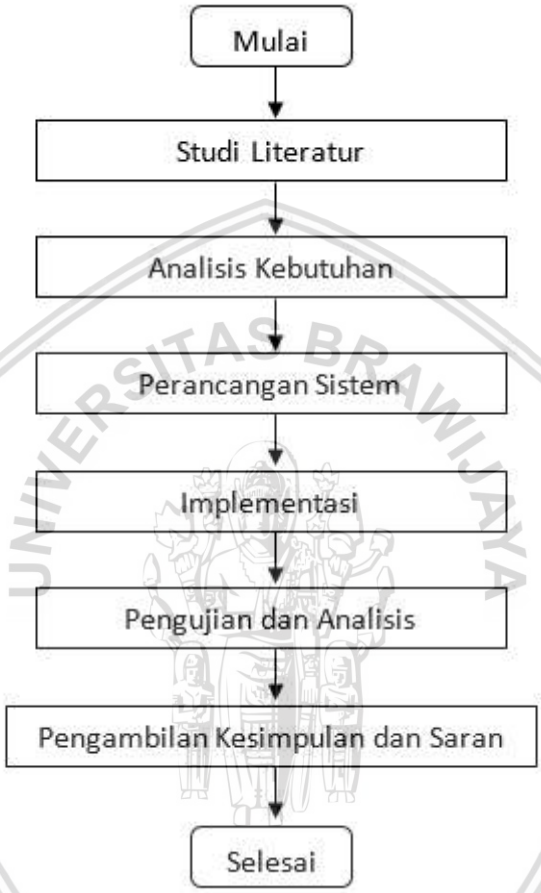
Gambar 2.6 sFlow-RT

Sumber: (InMon, 2017)

Secara kontinu mesin analitik pada sFlow-RT akan menerima aliran telemetri dari agen-agen sFlow yang terpasang pada perangkat-perangkat jaringan, *hosts* dan aplikasi-aplikasi. Aliran telemetri tersebut diubah menjadi metrik yang dapat diakses menggunakan REST API. Hal tersebut dapat mempermudah dalam melakukan konfigurasi terhadap pengukuran yang dapat disesuaikan, mendapatkan data metrik, mengatur *thresholds*, dan menerima notifikasi. Aplikasi yang akan memanfaatkan sFLow dapat dibuat secara eksternal menggunakan bahasa yang mendukung HTTP/REST, atau secara internal menggunakan JavaScript/ECMAScript (InMon, 2017).

BAB 3 METODOLOGI

Pada bab ini dijelaskan mengenai langkah dan metode dalam mengerjakan penelitian implementasi *multipath routing* berbasis algoritme DFS yang dimodifikasi. Langkah-langkah yang dilakukan ditunjukkan pada gambar 3.1



Gambar 3.1 Diagram Alir Metode Penelitian

Berdasarkan gambar 3.1 berikut adalah uraian langkah-langkah dalam mengerjakan penelitian ini.

1. Studi literatur penelitian-penelitian sebelumnya tentang *software-defined-networking,* OpenFlow, *multipath routing, load-balancing,* serta aplikasi-aplikasi dan penerapan SDN pada emulasi Mininet dengan Ryu *controller.*
2. Analisis kebutuhan sistem yang terdiri dari kebutuhan fungsional dan kebutuhan non-fungsional.
3. Perancangan sistem meliputi perancangan topologi yang akan digunakan dalam pengujian, algoritme pencarian jalur, algoritme pemilihan jalur, serta algoritme *load-balancing* yang akan diterapkan.
4. Implementasi sistem pada jaringan OpenFlow SDN.

5. Pengujian dan analisis kinerja *multipath routing* berbasis algoritme DFS yang dimodifikasi.
6. Pengambilan kesimpulan sesuai hasil pengujian dan analisis yang telah dilakukan.

3.2 Studi Literatur

Pada penelitian ini, dilakukan studi literatur sebagai dasar dan landasan dalam melakukan perancangan, implementasi dan pengujian antara lain.

1. Konsep dasar mengenai SDN dan OpenFlow serta aplikasi dan penerapannya.
2. *Multipath routing* dan *load-balancing* pada jaringan
3. Ryu sebagai *controller* pada SDN beserta aplikasi dan penerapannya.
4. Parameter-parameter yang dapat digunakan sebagai pengujian.

3.3 Analisis Kebutuhan

Pada tahap ini merupakan langkah menganalisis kebutuhan untuk mengidentifikasi kebutuhan sistem. Kebutuhan tersebut dikategorikan sebagai kebutuhan fungsional dan kebutuhan non-fungsional.

3.3.1 Kebutuhan Fungsional

Kebutuhan fungsional mendefinisikan layanan-layanan yang harus diberikan sistem. Kebutuhan tersebut adalah sebagai berikut.

1. Sistem dapat melakukan pencarian n-jalur (jika ada) dari suatu sumber ke tujuan pada topologi jaringan yang akan digunakan dalam *multipath routing*.
2. Sistem dapat menentukan jalur yang optimal dari keseluruhan jalur yang telah ditemukan berdasarkan *link-bandwidth* dan beban *traffic* dari suatu *switch.*
3. Sistem dapat membagi beban *(load-balancing)* berdasarkan jalur yang telah dipilih.

3.3.2 Kebutuhan Non-fungsional

Kebutuhan non-fungsional terdiri dari kebutuhan perangkat keras dan perangkat lunak yang digunakan untuk mendukung pengembangan sistem pada penelitian ini.

a. Kebutuhan perangkat keras

   PC/Laptop dengan spesifikasi:

   - CPU: Intel Core i5.
   - RAM: 8GB.
   - Harddisk: 1TB.

b. Kebutuhan perangkat lunak

- Ubuntu Linux 16.04 64-bit: merupakan sistem operasi yang digunakan.
- Mininet: untuk melakukan emulasi jaringan.
- Ryu: *controller* SDN yang digunakan.
- OpenvSwitch versi 2.5.2: sebagai virtual OpenFlow *switch.*
- sFlow-RT: untuk *monitoring traffic* jaringan.
- Ping, Iperf3, tcpdump: alat-alat yang digunakan untuk pengujian.

3.4 Perancangan Sistem

Perancangan sistem merupakan langkah-langkah dalam merancang suatu sistem berdasarkan analisis kebutuhan yang telah dilakukan. Untuk memenuhi kebutuhan yaitu menghasilkan sistem jaringan yang dapat melakukan pengiriman paket melalui beberapa jalur dengan *load-balancing*. Langkah-langkah perancang tergambar pada gambar 3.2

Gambar 3.2 Tahap Perancangan Sistem

Berdasarkan gambar 3.2 perancangan sistem terdiri dari lima tahap yang dijabarkan sebagai berikut.

1. Perancangan topologi, adalah proses untuk menentukan topologi yang akan digunakan sebagai pengujian pada sistem ini yaitu topologi yang memiliki kemampuan *multipath*
2. Perancangan algoritme pencarian jalur, merupakan proses dalam merancang algoritme yang digunakan dalam melakukan pencarian jalur pada topologi pengujian. Algoritme pencarian harus mampu menemukan jalur pada topologi jaringan.
3. Perancangan metrik penilaian jalur, adalah proses merancang metrik penilaian jalur untuk menentukan *cost* dari suatu jalur. Hal ini dilakukan untuk menentukan baik atau buruknya suatu jalur.

4. Perancangan algoritme *load-balancing*, adalah tahapan merancang algoritme yang melakukan pembagian beban dan mendistribusikannya melalui jalur-jalur yang telah dipilih.
5. Perancangan instalasi jalur adalah tahapan merancang algoritme instalasi jalur di tiap *switch*.

3.5 Implementasi

Pada tahap implementasi memuat langkah-langkah untuk membuat sistem yang telah dirancang. Tahap-tahap implementasi yang dilakukan yaitu:

1. Melakukan instalasi perangkat lunak *mininet*, Ryu, *sFlow-RT,* iperf3.
2. Pembangunan topologi pada linkungan simulasi SDN menggunakan *mininet.*
3. Pengembangan aplikasi *multipath routing* berbasis DFS dan Dijkstra menggunakan Ryu *controller.*

3.6 Pengujian dan Analisis

Pada pengujian dan analis dilakukan untuk menguji keberhasilan dan kinerja sistem dalam melakukan *multipath routing* pada jaringan OpenFlow. Berikut adalah pengujian yang dilakukan pada sistem.

1. Pengujian pencarian jalur *single-path* dan *multipath.*
2. Pengujian *respone time*, yaitu waktu yang dibutuhkan protokol *routing* untuk melakukan *routing* hingga berhasil mengirimkan paket pertama.
3. Pengujian jumlah iterasi, yaitu banyaknya jumlah iterasi yang diperlukan untuk menemukan n-jalur.
4. Pengujian *execution time,* yaitu waktu yang dibutuhkan aplikasi mulai dari pencarian jalur hingga berhasil menemukan n-jalur yang akan digunakan dalam *multipath routing.*
5. Pengujian *throughput,* untuk mengetahui kinerja jaringan
6. Pengujian *multipath,* untuk mengetaui apakah *flow* dapat didistribusikan setara di tiap jalur.

3.7 Pengambilan Kesimpulan dan Saran

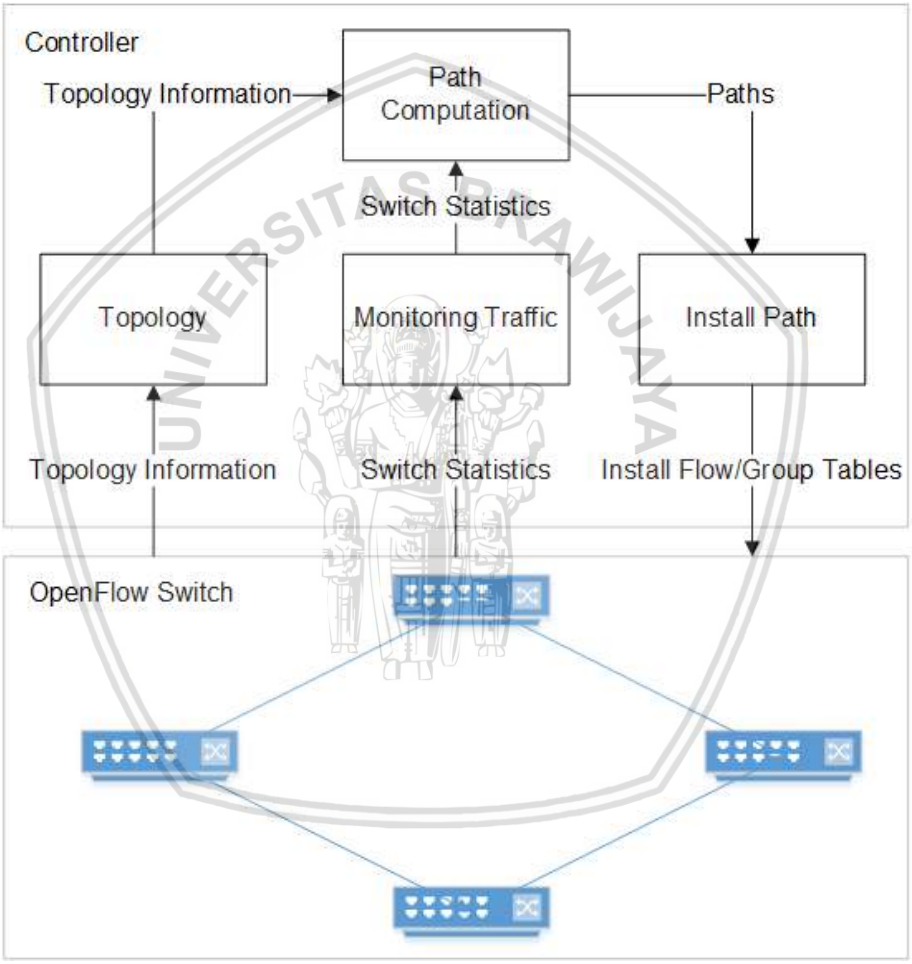
Pengambilan kesimpulan dan saran adalah langkah yang dilakukan setelah melakukan tahapan pengujian dan analisis pada *multipath routing* berbasis DFS dan Dijkstra pada jaringan OpenFlow. Berdasarkan hasil pengujian dapat ditentukan kinerja dari *multipath routing* dengan menggunakan algoritme DFS dan Dijkstra dalam pencarian rutenya. Pengambilan saran adalah tahap terakhir untuk mengevaluasi kesalahan-kesalahan sebagai pertimbangan pengembangan lebih lanjut.

BAB 4 PERANCANGAN DAN IMPLEMENTASI

Pada bab ini dijelaskan langkah-langkah dalam perancangan dan melakukan implementasi *multipath routing* berbasis DFS dan Dijkstra pada jaringan OpenFlow dengan mengikuti pedoman sesuai metodologi yang telah dibahas.

4.1 Perancangan Sistem

Pada tahap perancangan sistem akan dijelaskan proses-proses dalam merancang sistem yang akan dibangun berdasarkan spesifikasi kebutuhan yang telah dibahas.

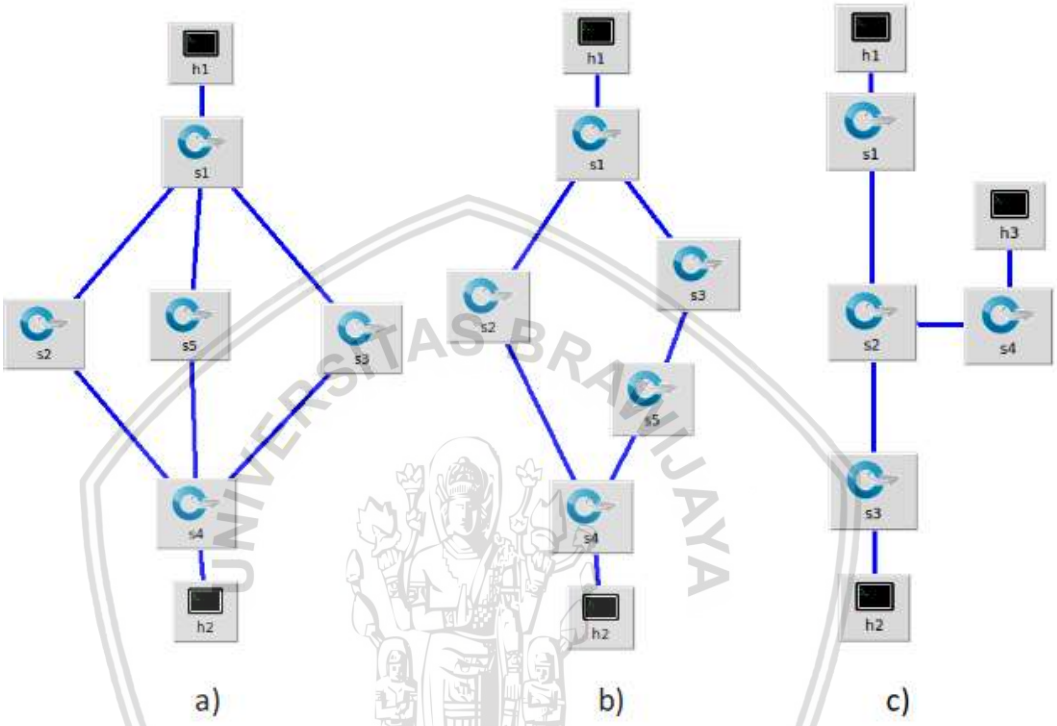


Gambar 4.1 Blok Diagram Sistem

Pada gambar 4.1 adalah blok diagram dari sistem. Modul *topology* mengidentifikasi *link* dan konektivitas pada OpenFLow *switch* sehingga *controller* dapat mengetahui informasi topologi. Modul *monitoring traffic* memberikan *controller* informasi *traffic* dari tiap *link* dan *switch.* Modul *path computation* akan menjalankan algoritme pencarian jalur dan menentukan n-jalur. Modul *install path* akan menginstall *flow table* atau *group table* berdasarkan jalur yang telah ditemukan.

4.1.1 Perancangan Topologi

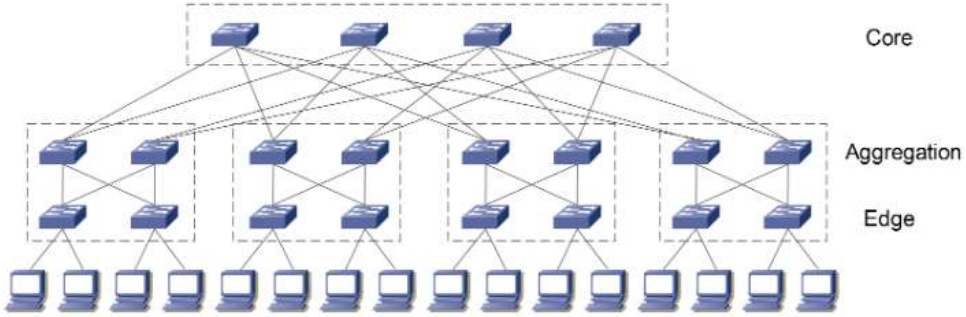
Untuk dapat memanfaatkan *multipath routing*, diperlukan suatu topologi yang memiliki beberapa jalur dari suatu sumber ke tujuan. Berikut adalah beberapa contoh topologi yang dapat digunakan dalam *multipath routing* dan yang tidak dapat digunakan dapat dilihat pada gambar 4.2



Gambar 4.2 Contoh Topologi Jaringan

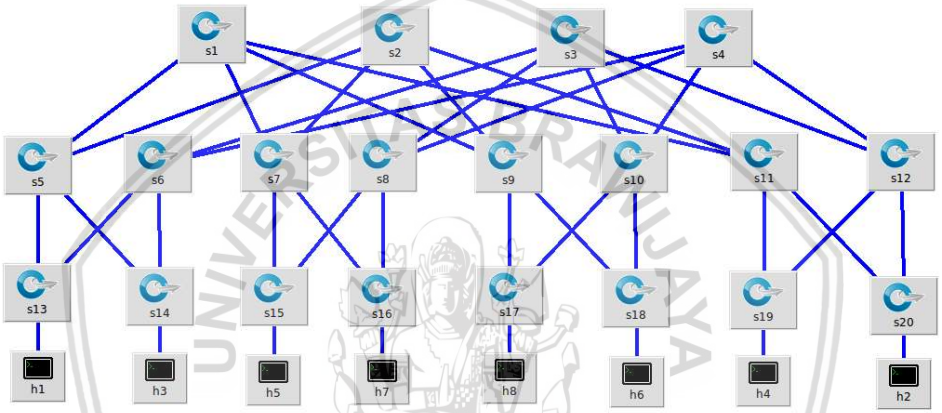
Pada gambar 4.2, topologi a) dan b) dapat digunakan dalam *multipath routing* karena memiliki beberapa jalur dari h1 dan h2. Sedangkan pada topologi c) hanya memiliki satu jalur antara h1 dan h2, h1 dan h2, h3 dan 1, maupun h3 dan h2. Oleh sebab itu topologi *tree* tidak cocok digunakan dalam *multipath routing.*

Untuk itu diperlukan suatu topologi pengujian yang memenuhi kriteria tersebut. Salah satu topologi yang memenuhi kriteria tersebut adalah topologi *fat-tree* yang sering diterapkan pada pusat data seperti pada gambar 4.3 dan dipetakan di Mininet pada gambar 4.4.



Gambar 4.3 Topologi *Fat Tree*

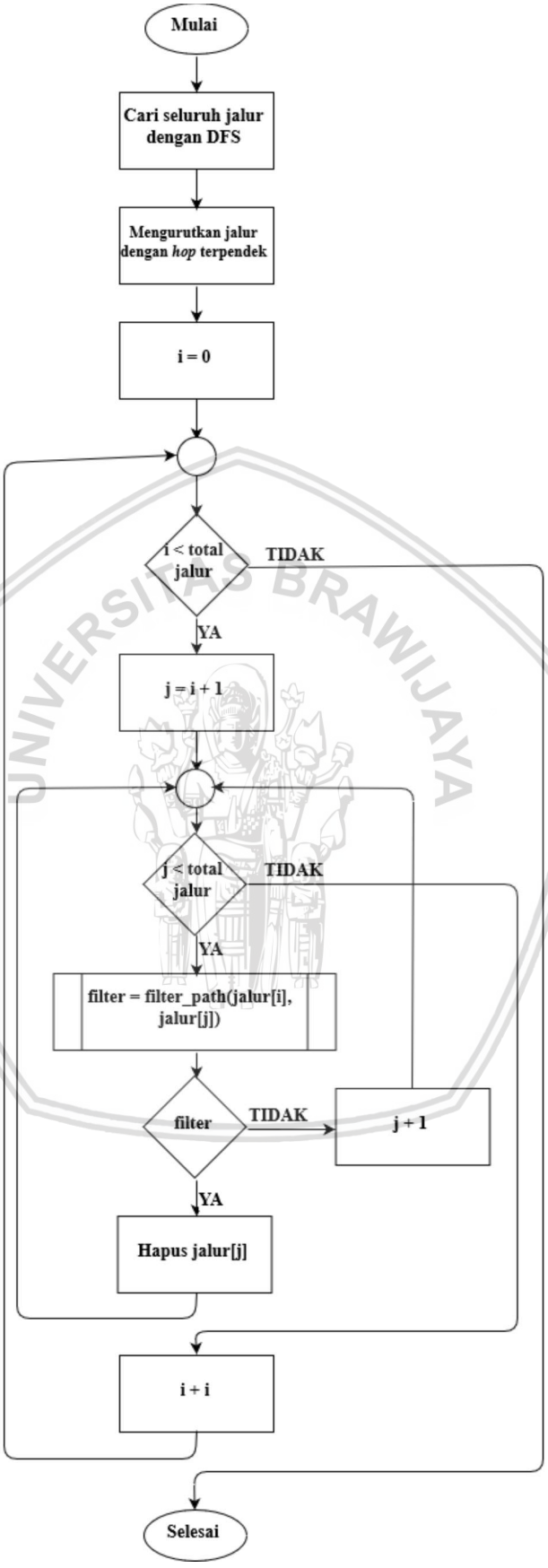
Sumber: (Wang, et al., 2014)



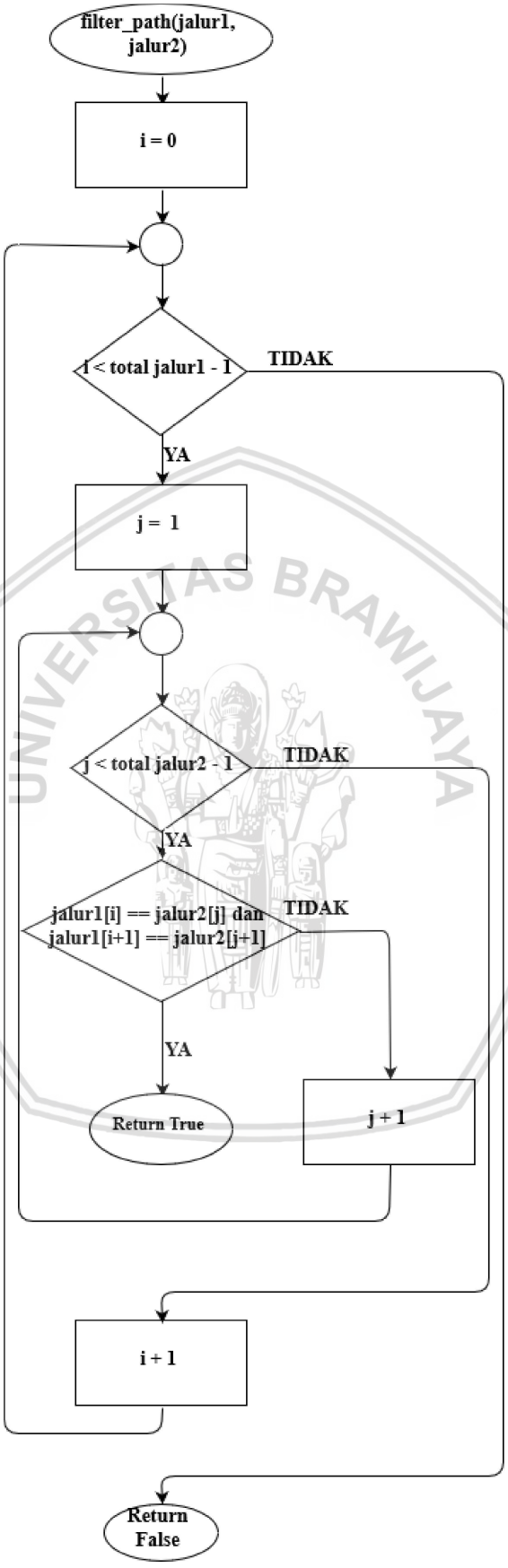
Gambar 4.4 Topologi *Fat-Tree* pada Mininet

4.1.2 Perancangan Algoritme Pencarian Jalur

Berdasarkan penelitan (Maulana, 2017) yang berjudul "*Multipath Routing* dengan *Load-Balancing* Pada *OpenFlow Software-Defined Network*" telah menerapkan *multipath routing* menggunakan algoritme DFS dengan memodifikasi sehingga algoritme pencarian jalur tersebut dapat menemukan keseluruhan jalur yang ada. Akan tetapi karena algoritme DFS mengekspansi *node* terjauh, hasil pencarian jalur yang ditemukan pertama kali adalah jalur terjauh dari sumber ke tujuan. Oleh sebab itu penulis mengembangkan algoritme tersebut sehingga dapat menemukan jalur terpendek dan menghapus jalur lain yang memiliki *shared edge* (*link* antara dua *node* yang ada pada jalur lain) sehingga menghasilkan jalur-jalur independen terdekat. Modifikasi algoritme DFS yang dilakukan adalah dengan mencari seluruh jalur dari suatu sumber ke tujuan, kemudian mengurutkan jalur dengan *hop* atau jarak terpendek. Setelah itu dilakukan seleksi terhadap jalur sehingga didapatkan jalur-jalur yang independen. Gambar 4.5 menunjukkan diagram alir algoritme DFS yang sudah dimodifikasi dan gambar 4.6 merupakan *pseudocode* dari algoritme DFS yang di modifikasi.



Gambar 4.5 Diagram Alir Modifikasi DFS

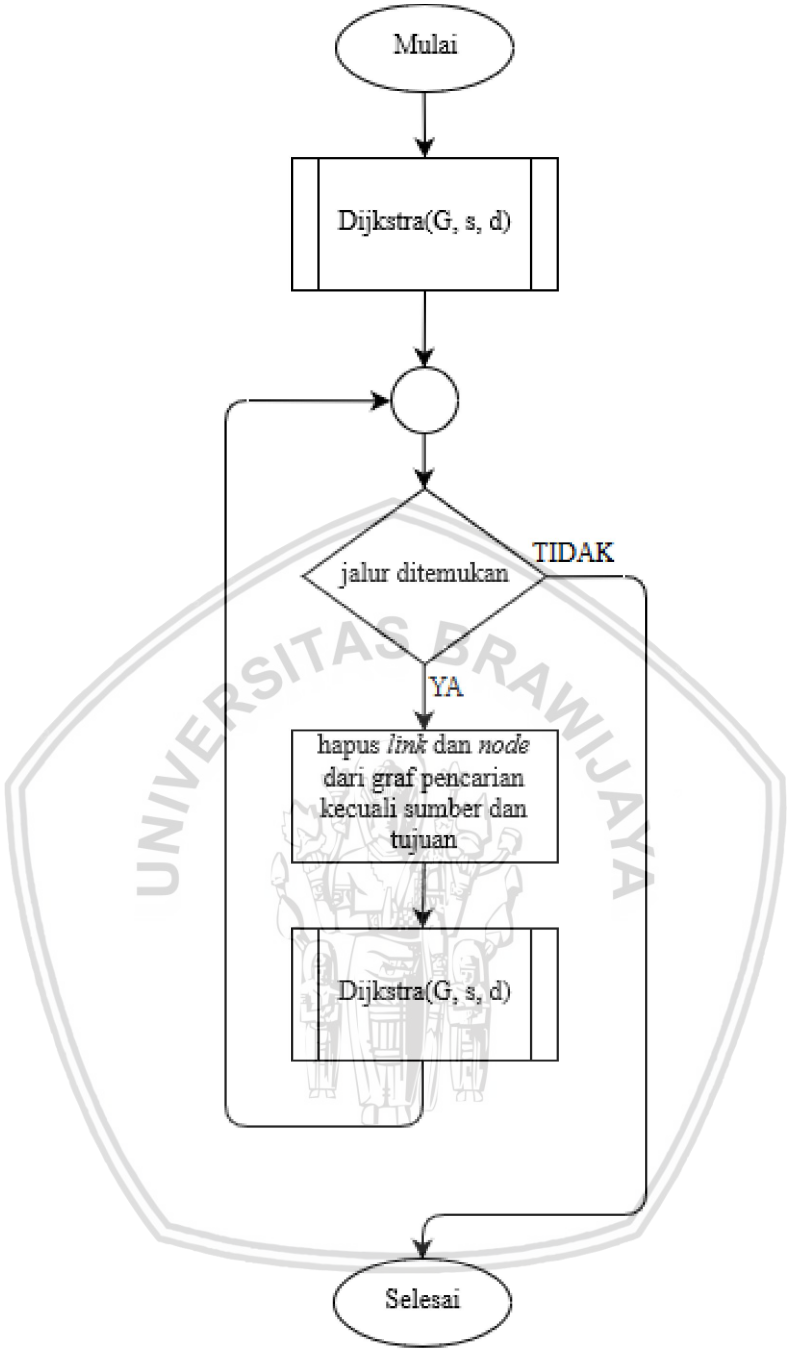


Gambar 4.6 Diagram Alir Modifikasi DFS (Lanjutan)

```
1   DFS(G, s, d)
2       p array to store paths
3       stack initialized as (s, [s])
4       while stack is not empty do
5           pop stack, save to node, p
6           for next in (G[node] - p)
7               if next is d
8                   append p + [next] to P
9               else
10                  push (next, p + [next]) to stack
11          end for
12      end while
13
14      sort element in p by length of path
15      i = 0
16      while i < length of p
17          j = i + 1
18          while j < length of p
19              if fileter_path(p[i], p[j])
20                  remove p[j] from p
21              else
22                  increment j
23          increment i
24      return p
25  end
26
27  filter_path(path1, path2)
28      for i in range(length of path1 - 1)
29          for j in range(length of path2 -1)
30              if path1[i] == path2[j] and path1[i+1] == path2[j+1]
31                  return True
32      return False
33  end
```

Gambar 4.7 Modifikasi Algoritme DFS

Algoritme pencarian berikutnya adalah Dijkstra. Pada algoritme Dijkstra seperti yang ditunjukkan gambar 2.4, pencarian jalur dari sumber ke tujuan hanya menghasilkan satu jalur terpendek. Berdasarkan penelitian yang dilakukan oleh (Chiang, et al., 2017) yang berjudul "*A Multipath Transmission Scheme for the Improvement of Throughput over SDN*" telah menerapkan *multipath routing* menggunakan algoritme Dijkstra dengan melakukan iterasi pencarian jalur pada graf topologi jaringan. Pada tiap iterasi akan dihasilkan satu jalur terpendek dari sumber ke tujuan, kemudian jalur tersebut di hapus dari graf pencarian dengan menghapus *link* dan *node* pada graf pencarian kecuali *node* sumber dan *node* tujuan. Gambar 4.8 menunjukkan diagram alir algoritme Dijkstra yang dimodifikasi dan gambar 4.9 merupakan *pseudocode* dari algoritme Dijkstra yang dimodifikasi.



Gambar 4.8 Diagram Alir Modifikasi Algoritme Dijkstra

```
1   find_allpaths
2      path = Dijkstra(Graph, src, dst)
3      while path != []
4         remove all links and nodes from Graph except src and dst
5         add path to all_paths
6         path = Dijkstra(Graph, source, dst)
7      return all_paths
8   End
```

Gambar 4.9 Modifikasi Algoritme Dijkstra

4.1.3 Perancangan Metrik Penilaian Jalur

Pada protokol OSPF, perhitungan *link cost* menggunakan rumus sebagai berikut (Cisco, 2005).

$$OSPF\ Cost = \frac{B_R}{B_L} \quad (4.1)$$

Pada rumus 4.1, B_R adalah *reference bandwidth* yaitu sebesar 100 Mbps dan B_L adalah *link bandwidth* antara sepasang *router* yang terhubung. Namun rumus OSPF tersebut tidak memperhitungkan beban *traffic* pada *switch.* SDN dapat mengatasi permasalahan tersebut dengan menyediakan informasi statistik terhadap *flow* pada suatu *switch.* Penelitian yang dilakukan (Jiang, et al., 2014) telah mengusulkan ekstensi terhadap algoritme Dijkstra dengan melakukan penambahan penilaian *node weight* dan *edge weight* pada pencarian rutenya. Penilaian *node weight* didefinisikan pada persamaan 4.2 dan *edge weight* pada persamaan 4.3.

$$nw(v) = \frac{\sum_{f \in Flow(v)} Bits(f)}{Capacity(v)} \quad (4.2)$$

$$ew(e) = \frac{\sum_{f \in Flow(e)} Bits(f)}{Bandwidth(e)} \quad (4.3)$$

Diasumsikan sebuah graf G= (V, E) yang diturunkan dari topologi SDN, yang memiliki bobot, berarah, dan terhubung. Untuk sebuah *node* $v \in V$ dan *edge* $e \in E$, *Flow(v),* dan *Flow(e)* adalah sekumpulan dari semua *flow* yang melewati *v* dan *e*, *Capacity(v)* merupakan kapasitas dari *v* (banyaknya *bits* yang dapat diproses *v* dalam satu detik), dan *Bandwidth(e)* merupakan *bandwidth* dari *e* (banyaknya *bits* yang dapat dikirimkan oleh *e* dalam satu detik).

Dengan persamaan tersebut dapat ditentukan matrik penilaian jalur sebagai total *edge weigth* seperti pada persamaan 4.4.

$$pw(p) = \sum_{e \in E} ew(e) \quad (4.4)$$

Pada persamaan (4.4), *pw(p)* menyatakan bobot suatu jalur (*path weight), p (V, E)* adalah jalur yang terdiri dari sekumpulan *edge*. Bobot tersebut dihitung dengan menjumlahkan seluruh *edge weight* pada suatu jalur. Jalur terbaik adalah jalur dengan nilai *pw(p)* paling rendah.

Penelitian yang dilakukan (Maulana, 2017) menentukan pemilihan jalur berdasarkan nilai dari *pw* dengan cara mengurutkan jalur berdasarkan *pw* paling rendah. Jalur yang memiliki nilai *pw* paling rendah akan dipilih dan digunakan dalam *multipath routing*. Untuk perhitungan bobot menggunakan persentase dari *pw* tiap-tiap jalur dengan keseluruhan *pw* jalur seperti pada persamaan 4.5.

$$w(p) = \left(1 - \frac{pw(p)}{\sum_{i<n}^{i=0} pw(i)}\right) x\ 10 \quad (4.6)$$

4.1.4 Perancangan Algoritme *Load-Balancing*

Pada proses *load-balancing* dibutuhkan mekanisme yang cepat dan tetap memperhitungkan bobot dari jalur yang digunakan. Untuk itu, proses *load-balancing* dilakukan dengan menggunakan mekanisme yang telah disediakan oleh *OpenvSwitch* dengan memanfaatkan *group table* dan *bucket action* dari spesifikasi OpenFlow 1.3 (Open Networking Foundation, 2012). Proses *load-balancing* dilakukan oleh *switch* tanpa harus berkonsultasi dengan *controller* dengan cara *hashing* terhadap *header packet layer 2* hingga *layer 4* seperti MAC, IP, dan TCP *port* yang dikalikan dengan bobot dari jalur.

4.1.5 Perancangan Instalasi Jalur

Setelah didapatkan jalur-jalur dari suatu sumber ke tujuan, diperlukan proses instalasi jalur tersebut pada tiap *switch*.

```
1  install_paths()
2     list path from source to destination
3     for switch in path do
4        list all port in switch that contain path
5        if port > 1
6           install group table
7        else
8           install normal flow table
```

Gambar 4.10 Algoritme Instalasi Jalur

4.2 Implementasi

Pada tahap implementasi akan dijelaskan langkah-langkah dalam mengimplementasikan sistem sesuai dengan metodologi penelitian. Berikut adalah perangkat lunak pendukung dan langkah-langkah instalasinya.

4.2.1 Instalasi

4.2.1.1 Mininet

Mininet adalah perangkat lunak yang merupakan *emulator* jaringan yang digunakan untuk pengembangan sistem pada penelitian ini. Langkah-langkah dalam melakukan instalasi Mininet adalah sebagai berikut.

1. Megunduh kode sumber *Mininet* dari github resmi *Mininet* dengan perintah:

   $ git clone git://github.com/mininet/mininet.git

2. Melakukan instalasi *Mininet* dengan perintah:

   $ sudo mininet/util/install.sh -a

4.2.1.2 OpenvSwitch

Dalam pengembangan sistem ini menggunakan *OpenvSwitch* versi 2.5.2. Pada sistem operasi Ubuntu 16.04, *OpenvSwitch* versi 2.5.2 sudah ada dalam *repository*

Ubuntu sehingga saat melakukan instalasi *Mininet*, *OpenvSwitch* versi 2.5.2 otomatis terinstal.

4.2.1.3 Ryu *Controller*

Ryu merupakan *controller* SDN yang digunakan dalam pengembangan sistem pada penelitan ini. Berikut adalah langkah-langkah instalasi Ryu.

1. Mengunduh kode sumber Ryu dari github resmi Ryu dengan perintah:

   $ git clone git://github.com/osrg/ryu.git

2. Memindah direktori ke "ryu" dengan perintah:

   $ cd ryu

3. Mengistal *package depedencies* ryu dengan perintah:

   $ sudo pip install -r tools/pip-requires

4. Mengistal *optional dependencies* ryu dengan perintah:

   $ sudo pip install -r tools/optional-requires

5. Mengistal ryu dengan perintah:

   $ sudo python ./setup.py install

4.2.1.4 *sFlow-RT*

Berikut adalah langkah-langkah instalasi *sFlow-RT* sebagai monitor jaringan.

1. Mengunduh *tarball sFlow-RT* dari *repository* resmi dengan perintah:

   $ wget http://www.inmon.com/produc ts/sFlow-RT/sflow-rt.tar.gz

2. Mengekstrak *tarball sFlow-RT* yang telah diunduh dengan perintah:

   $ tar xvzf sflow-rt.tar.gz

3. Melakukan instalasi java dengan perintah:

   $ sudo apt install default-jre

4.2.1.5 Iperf3

Berikut adalah langkah-langkah instalasi iperf3 sebagai alat dalam pengujian *throughput*.

1. Memperbarui repositori dengan perintah:

   $ sudo apt-get update

2. Mengistal iperf3 dengan perintah:

   $ sudo apt-get install iperf3

4.2.2 Membangun Topologi di *Mininet*

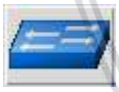
Langkah berikutnya setelah tahap instalasi perangkat lunak pendukung selesai, adalah membuat topologi di *Mininet* sesuai perancangan topologi yang telah dilakukan. Dalam membangun topologi di *Mininet* dapat menggunakan *GUI (Graphical User Interface)* yang disediakan oleh *Mininet* yaitu *Miniedit* dengan langkah sebagai berikut.

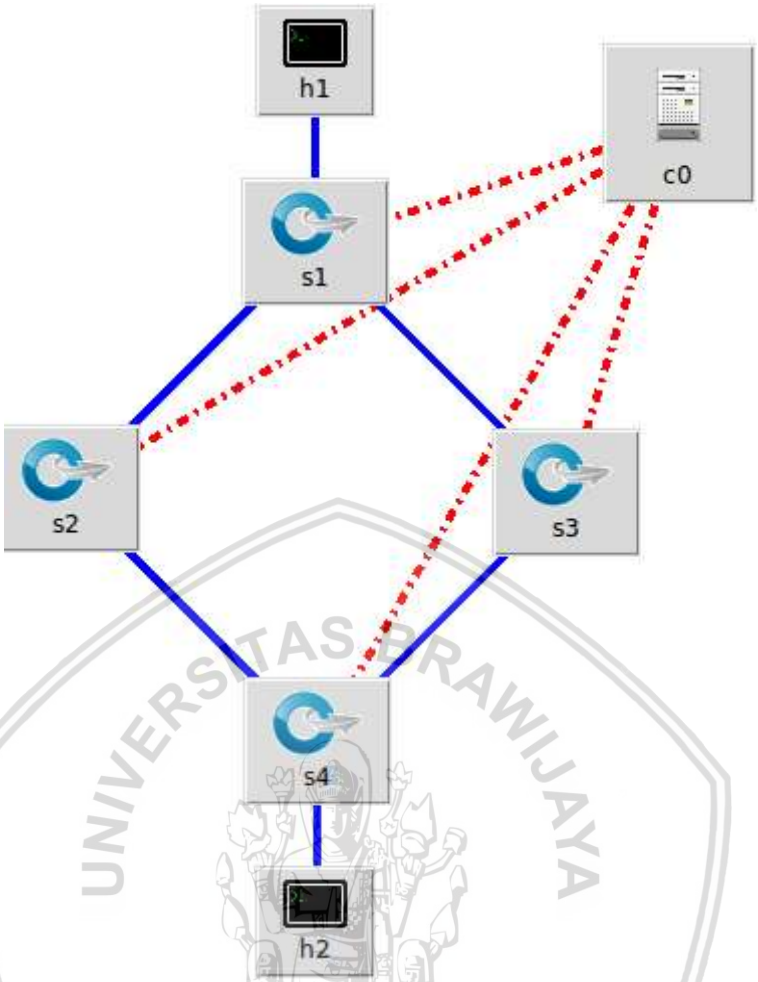
1. Menjalankan *Miniedit* melalui terminal dengan perintah sebagai berikut:

   $ sudo python mininet/examples/minedit.py

2. Membuat topologi dengan mengacu pada komponen-komponen yang disediakan oleh Miniedit yang ditunjukkan tabel 4.1. Contoh hasil pembuatan topologi menggunakan Miniedit dapat dilihat pada gambar 4.3

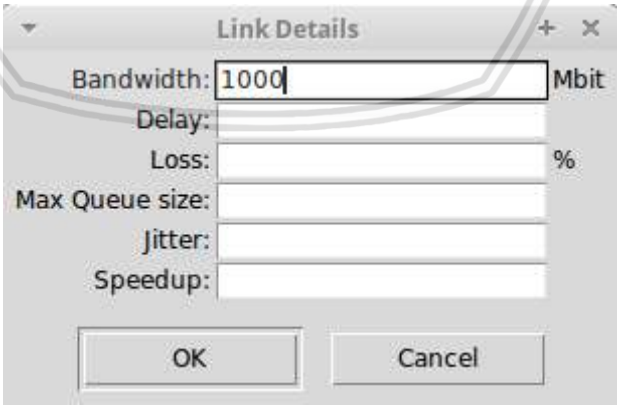
Tabel 4.1 Daftar Komponen Miniedit

No	Ikon	Nama	Fungsi
1		*Select*	Digunakan untuk memindahkan dan memilih suatu komponen
2		*Host*	Komponen *host* atau *end device* yang berfungsi seperti sebuah komputer
3		*Switch*	Perangkat jaringan *OpenFlow* yang bertugas melakukan *forwarding (data plane)* pada SDN
4		*LegacySwitch*	Perangkat jaringan *switch* tradisional yang bertugas meneruskan paket dengan cara mempelajari alamat MAC berdasarkan letak *port* di *switch.*
5		*LegacyRouter*	Perangkat jaringan *router* tradisional yang meneruskan paket berdasarkan protokol *routing* tertentu seperti OSPF dan RIP.
6		*NetLink*	Digunakan untuk menghubungkan satu komponen dengan komponen lain
7		*Controller*	Perangkat *controller* SDN yang diakses *Mininet* pada alamat dan *port* tertentu.



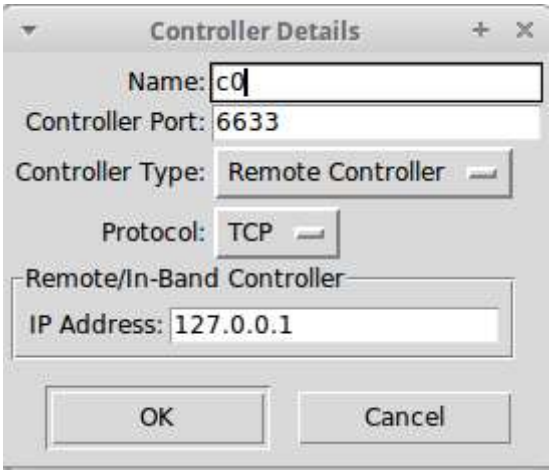
Gambar 4.11 Contoh Pembangunan Topologi di Miniedit

3. Melakukan konfigurasi *bandwidth* di tiap *link* dengan cari klik kanan pada *link* kemudian *properties* lalu mengatur *bandwidth* pada *field bandwidth*



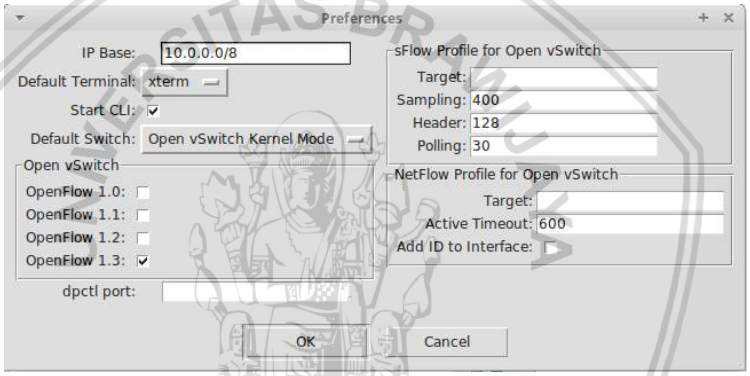
Gambar 4.12 Pengaturan *Link Bandwidth*

4. Melakukan konfigurasi pada *controller* seperti pada gambar 4.5 dengan cara klik kanan pada *controller* (c0), kemudian klik *properties.* Pada bagian "*Controller Type"* menggunakan "*Remote Controller".* Hal ini dilakukan agar *controller* dapat terhubung dengan Ryu *controller.*



Gambar 4.13 Pengaturan *Controller Pada Miniedit*

5. Melakukan konfigurasi *preferences* pada *Miniedit* sesuai gambar 4.6 dengan cara klik pada menu "Edit" lalu memilih "*Preferences".*



Gambar 4.14 Pengaturan *Preferences* Pada *Miniedit*

4.2.3 Langkah - Langkah Menjalankan Sistem

1. Menjalankan simulasi topologi di *Mininet* dengan cara klik "*Run"* pada *Miniedit.*
2. Membuka terminal baru dan menjalankan *sFlow* dengan cara berpindah menuju direktori *sFlow* dan menjalankan "*start.sh"* menggunakan perintah berikut:

   $ cd sflow-rt

   $ ./start.sh
3. Membuka terminal baru kemudian menjalankan program *controller* menggunakan *ryu-manager* dengan perintah:

   $ sudo ryu-manager --observe-links program.py

4.2.4 Pengembangan Program *Controller*

Langkah pengembangan program *controller* dengan Ryu, terdiri dari pemrograman dengan menggunakan bahasa Pyhton versi 2.7.12 dalam mengimplementasikan logika program sesuai perancangan.

4.2.4.1 Variabel dan Struktur Data

Berikut ini adalah beberapa variabel dan struktur data pada program yang ditunjukkan pada tabel 4.2.

Tabel 4.2 Variabel dan Struktur Data

No	Nama Variabel	Tipe Data	Deskripsi	Contoh
1	*thr*	*Dictionary*	Digunakan untuk menyimpan informasi beban *traffic* dari suatu *switch*	{1:{1:5234.2423,2:0}} Penjelasan: *switch* 1 pada *port* 1 memiliki beban *traffic* 5234.2423 *bps*. Pada port 2 beban *traffic* 0 *bps*
2	*switches*	*Dictionary*	Berisi informasi pada *switch*	{1:{"capacity":10000000000, "ports":{1:{"ifindex":47, "ifname":"s1-eth1", "bandwidth":100000000}}}} Penjelasan: *switch* 1 memiliki kapasitas 10*Gbps*. Pada *port* 1 memiliki *bandwidth* 100*Mbps*
3	*mymac*	*Dictionary*	Digunakan untuk menyimpan informasi pemetaan alamat MAC pada *switch* dan *port*-nya	{"0A:F3:D1:0A:41:3E": (1,3)} Penjelasan: alamat MAC "0A:F3:D1:0A:41:3E" terhubung pada *switch* 1 di port 3
4	*adjacency*	*Dictionary*	Menyimpan informasi ketetanggaan antara dua switch	{2:{3:1,4:2}}

Tabel 4.3 Variabel dan Struktur Data (Lanjutan)

No	Nama Variabel	Tipe Data	Deskripsi	Contoh
5	*MAX_PATH*	*Integer*	Jumlah maksimal jalur yang akan digunakan dalam *multipath routing*	2 Penjelasan: 2 jalur
6	DFS	*Boolean*	Algoritme pencarian jalur yang digunakan	True Penjelasan: menggunakan algoritme DFS

4.2.4.2 Kode Sumber Pencarian Jalur

Berikut adalah implementasi algoritme pencarian jalur sesuai dengan perancangan algoritme pencarian jalur pada sub bab 4.1.2. Terdapat dua algoritme, yaitu DFS dan Dijkstra. Algoritme yang akan dijalankan tergantung pada nilai dari variabel DFS. Jika variabel DFS bernilai *True* maka algoritme yang digunakan adalah DFS, dan *False* untuk algoritme Dijkstra

Tabel 4.4 Kode Sumber Algoritme Pencarian Jalur

```
1   def dfs(src, dst):
2       paths = []
3       stack = [(src, [src])]
4       while stack:
5           (node, path) = stack.pop()
6           for next in set(adjacency[node].keys()) - set(path):
7               if next is dst:
8                   paths.append(path + [next])
9               else:
10                  stack.append((next, path + [next]))
11
12      print "All paths from ", src, " to ", dst, " : ", paths
13      paths.sort(key=len)
14      print "Sorted paths by length from ", src, " to ", dst, " : ",
15  paths
16
17      i = 0
18      paths_del = []
19      while i < len(paths):
20          j = i + 1
21          while j < len(paths):
22              if filter_path(paths[i], paths[j]):
23                  paths_del.append(paths[j])
24                  paths.remove(paths[j])
25              else:
26                  j += 1
27          i += 1
28
29      print "Removed paths from ", src, " to ", dst, " : ", paths_del
30      print "Available paths from ", src, " to ", dst, " : ", paths
31      return paths
32
33
```

Tabel 4.5 Kode Sumber Algoritme Pencarian Jalur (Lanjutan)

```
34  def minimum_distance(distance, Q):
35    min = float('Inf')
36    node = 0
37    for v in Q:
38      if distance[v] < min:
39        min = distance[v]
40        node = v
41    return node
42
43
44  def dijkstra(graph, src, dst):
45      # Dijkstra's algorithm
46      print "get_path is called, src=", src, " dst=", dst
47      if src not in graph.keys():
48          return []
49      if dst not in graph.keys():
50          return []
51
52      distance = {}
53      previous = {}
54
55      for dpid in graph.keys():
56          distance[dpid] = float('Inf')
57          previous[dpid] = None
58
59      distance[src] = 0
60      Q = set(graph.keys())
61      print "Q=", Q
62
63      while len(Q) > 0:
64          u = minimum_distance(distance, Q)
65          Q.remove(u)
66
67          for p in graph.keys():
68              if graph[u].has_key(p):
69                  w = 1
70                  if distance[u] + w < distance[p]:
71                      distance[p] = distance[u] + w
72                      previous[p] = u
73
74      r = []
75      p = dst
76      r.append(p)
77      q = previous[p]
78      while q is not None:
79          if q == src:
80              r.append(q)
81              break
82          p = q
83          r.append(p)
84          q = previous[p]
85
86      r.reverse()
87      if src == dst:
88          path = [src]
89      else:
90          path = r
91
92      return path
```

Tabel 4.6 Kode Sumber Algoritme Pencarian Jalur (Lanjutan)

```
93  def get_paths(src, dst):
94      paths = []
95      if(DFS):
96          paths = dfs(src,dst)
97      else:
98          graph = copy.deepcopy(adjacency)  # Copy variable adjacency
99  to graph
100
101         path = dijkstra(graph, src, dst)
102
103         while path:
104             for i in range(len(path) - 1):
105                 del graph[path[i]][path[i + 1]]
106                 del graph[path[i + 1]][path[i]]
107
108             for i in range(len(path)):
109                 if not graph[path[i]]:
110                     del graph[path[i]]
111
112             path = dijkstra(graph, src, dst)
113
114         print "Available paths from ", src, " to ", dst, " : ",
115 paths
116
117     return  paths
```

4.2.4.3 Kode Sumber Penilaian Jalur

Setelah jalur antara sumber dan tujuan ditemukan, dilakukan penilaian jalur sesuai sub bab 4.13. Berikut adalah implementasi algoritme penilaian jalur

Tabel 4.7 Kode Sumber Algoritme Penilaian Jalur

```
1   def get_link_cost(s1, s2):
2       e1 = adjacency[s1][s2]
3       e2 = adjacency[s2][s1]
4       bl = min(switches[s1]['ports'][e1]['bandwidth'],
5                switches[s2]['ports'][e2]['bandwidth'])
6       if bl == 0 :
7           ew = 1
8       else:
9           ew = ((thr[s2][e1]/bl)) + 1
10      return ew
11
12
13  def get_path_cost(path):
14      cost = 0
15      for i in range(len(path) - 1):
16          cost += get_link_cost(path[i], path[i + 1])
17      return cost
```

4.2.4.4 Kode Sumber Pemilihan Jalur

Setelah dilakukan penilaian jalur, dapat ditentukan beberapa jalur terbaik berdasarkan nilai variabel *MAX_PATHS.*

Tabel 4.8 Kode Sumber Algoritme Pemilihan Jalur

```
def get_n_paths(src, dst):
    '''
    Get the n paths according to MAX_PATHS
    '''
    paths = get_paths(src, dst)
    #paths = paths.sort(key=len)
    paths_count = len(paths) if len(
        paths) < MAX_PATHS else MAX_PATHS
    return          sorted(paths,          key=lambda          x:
get_path_cost(x))[0:(paths_count)]
```

4.2.4.5 Kode Sumber *Monitoring Traffic*

Untuk menghitung *link cost* atau *edge weight* dibutuhkan informasi beban *traffic* dari suatu *link* yang dapat diambil dari *sFlow-RT.*

Tabel 4.9 Kode Sumber *Monitoring Traffic*

```
def monitor_traffic():
    '''
    Measure outgoing traffic per second for all switch ports
    '''
    while True:
        try:
            a = switches
            for switch in switches:
                for port in switches[switch]['ports']:
                    url = 'http://' + collector + ':8008/metric/' +
\
                          collector         +          '/'          +
switches[switch]['ports'][port]['ifindex'] + \
                          '.ifoutoctets/json'
                    r = get(url)
                    response = r.json()
                    # print response
                    try:
                        # Bps to bps
                        thr[switch][port]                          =
response[0]['metricValue'] * 8
                    except KeyError:
                        pass
                    # print switch,thr[switch]
        except RuntimeError:
            pass
        hub.sleep(1)
```

4.2.4.6 Kode Sumber Instalasi Jalur

Setelah ditemukan jalur dari sumber ke tujuan dilakukan instalasi jalur tersebut pada masing-masing *switch*

Tabel 4.10 Kode Sumber Instalasi Jalur

```
def install_paths(self, src, first_port, dst, last_port, ip_src,
ip_dst):
   computation_start = time.time()
   paths = self.get_n_paths(src, dst)
   pw = []
   for path in paths:
       pw.append(self.get_path_cost(path))
       print path, "cost = ", pw[len(pw) - 1]
   sum_of_pw = sum(pw)
   paths_with_ports  =  self.add_ports_to_paths(paths,  first_port,
last_port)
   switches_in_paths = set().union(*paths)

   for node in switches_in_paths:

       dp = self.datapath_list[node]
       ofp = dp.ofproto
       ofp_parser = dp.ofproto_parser
       ports = defaultdict(list)
       actions = []
       i = 0

       for path in paths_with_ports:
           if node in path:
               in_port = path[node][0]
               out_port = path[node][1]
               if (out_port, pw[i]) not in ports[in_port]:
                   ports[in_port].append((out_port, pw[i]))
           i += 1

       for in_port in ports:

           match_ip = ofp_parser.OFPMatch(
               eth_type=0x0800,
               ipv4_src=ip_src,
               ipv4_dst=ip_dst
           )
           match_arp = ofp_parser.OFPMatch(
               eth_type=0x0806,
               arp_spa=ip_src,
               arp_tpa=ip_dst
           )

           out_ports = ports[in_port]
           #print out_ports

           if len(out_ports) > 1:
               group_id = None
               group_new = False

               if (node, src, dst) not in self.multipath_group_ids:
                   group_new = True
                   self.multipath_group_ids[
                       node,          src,          dst]          =
self.generate_openflow_gid()
               group_id = self.multipath_group_ids[node, src, dst]

               buckets = []
               # print "node at ",node," out ports : ",out_ports
               for port, weight in out_ports:
                   bucket_weight = int(round((1 - weight/sum_of_pw)
* 10))

```

Tabel 4.11 Kode Sumber Instalasi Jalur (Lanjutan)

```
                    bucket_action                                   =
[ofp_parser.OFPActionOutput(port)]
                    buckets.append(
                        ofp_parser.OFPBucket(
                            weight=bucket_weight,
                            watch_port=port,
                            watch_group=ofp.OFPG_ANY,
                            actions=bucket_action
                        )
                    )

                if group_new:
                    req = ofp_parser.OFPGroupMod(
                        dp,     ofp.OFPGC_ADD,     ofp.OFPGT_SELECT,
group_id,
                        buckets
                    )
                    dp.send_msg(req)
                else:
                    req = ofp_parser.OFPGroupMod(
                        dp, ofp.OFPGC_MODIFY, ofp.OFPGT_SELECT,
                        group_id, buckets)
                    dp.send_msg(req)

                actions = [ofp_parser.OFPActionGroup(group_id)]

                self.add_flow(dp, 32768, match_ip, actions)
                self.add_flow(dp, 1, match_arp, actions)

            elif len(out_ports) == 1:
                actions                                             =
[ofp_parser.OFPActionOutput(out_ports[0][0])]

                self.add_flow(dp, 32768, match_ip, actions)
                self.add_flow(dp, 1, match_arp, actions)
        print "Path installation finished in ", time.time()  -
computation_start
    return paths_with_ports[0][src][1]
```

BAB 5 PENGUJIAN DAN ANALISIS

Pada bab ini dijelaskan hasil pengujian dan analisis dari *multipath routing* berbasis algoritme DFS yang dimodifikasi

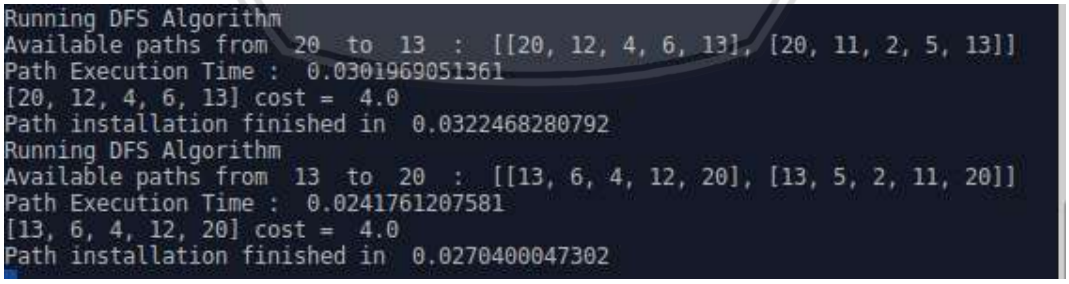
5.1 Pengujian Pencarian Jalur

Pengujian pencarian jalur dilakukan berdasarkan perancang topologi yang telah dilakukan yaitu pada gambar 4.3 dengan mengamati jalur yang ditemukan. Pada topologi tersebut menggunakan *link bandwidth* yang digunakan pada topologi tersebut adalah 1000 *Mbps*

Terdapat beberapa skenario yang dilakukan yaitu pencarian jalur *single-path* dan *multipath* dengan menggunakan algoritme DFS dan Dijkstra. Pada pencarian *single-path* variabel *MAX_PATH* diubah menjadi satu. Untuk melakukan pencarian jalur digunakan aplikasi *ping* dari sumber h1 ke tujuan h2.

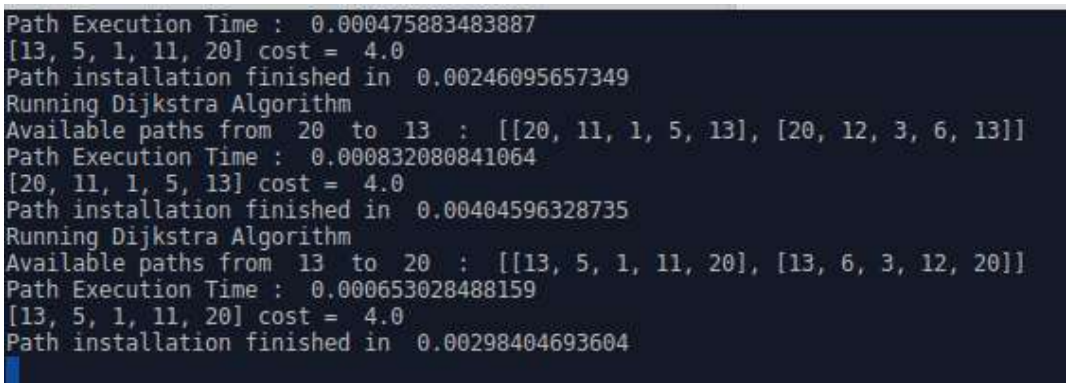
Gambar 5.1 Hasil Pencarian Jalur *Single*-path Menggunakan DFS

Pada gambar 5.1 merupakan hasil pencarian jalur *single-path* antara h1 dan h2 menggunakan algoritme DFS yang tidak modifikasi.



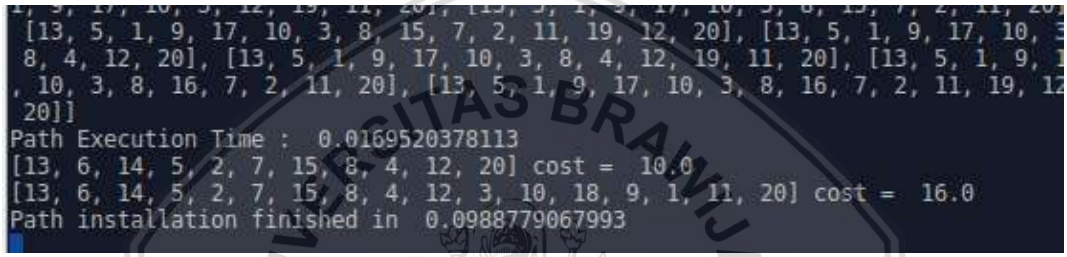
Gambar 5.2 Hasil Pencarian Jalur *Single*-path Menggunakan *Modified* DFS

Pada gambar 5.2 merupakan hasil pencarian jalur *single-path* antara h1 dan h2 dengan menggunakan algoritme DFS yang sudah di modifikasi.



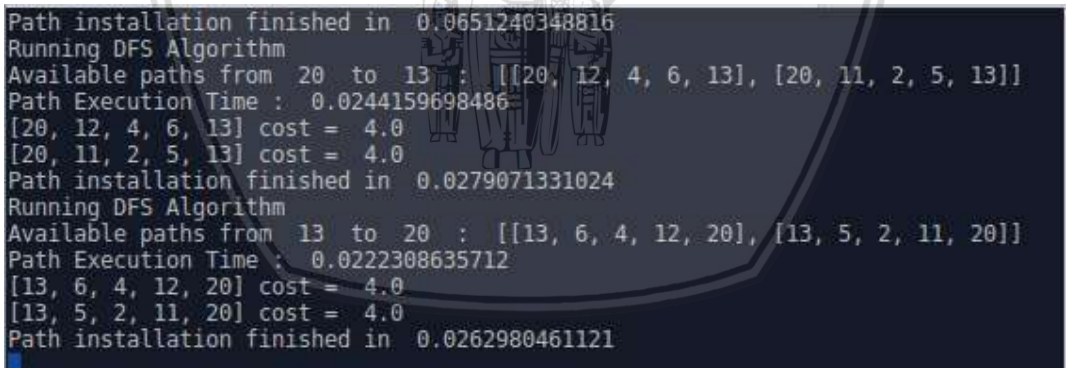
Gambar 5.3 Hasil Pencarian Jalur *Single*-path Menggunakan Dijkstra

Pada gambar 5.3 adalah hasil pencarian jalur *single-path* antara h1 dan h2 menggunakan algoritme Dijkstra.



Gambar 5.4 Hasil Pencarian Jalur *Multipath* Menggunakan DFS

Pada gambar 5.4 adalah hasil pencarian jalur *multipath* antara h1 dan h2 menggunakan DFS.



Gambar 5.5 Hasil Pencarian Jalur *Multipath* Menggunakan *Modified* DFS

Pada gambar 5.5 adalah hasil pencarian jalur *multipath* antara h1 dan h2 menggunakan DFS yang dimodifikasi.

```
Path installation finished in  0.00690913200378
Running Dijkstra Algorithm
Available paths from  20  to  13  :  [[20, 11, 1, 5, 13], [20, 12, 3, 6, 13]]
Path Execution Time :  0.000655174255371
[20, 11, 1, 5, 13] cost =  4.0
[20, 12, 3, 6, 13] cost =  4.0
Path installation finished in  0.00402998924255
Running Dijkstra Algorithm
Available paths from  13  to  20  :  [[13, 5, 1, 11, 20], [13, 6, 3, 12, 20]]
Path Execution Time :  0.00040602684021
[13, 5, 1, 11, 20] cost =  4.0
[13, 6, 3, 12, 20] cost =  4.0
Path installation finished in  0.0954999923706
```

Gambar 5.6 Hasil Pencarian Jalur *Multipath* Menggunakan Dijkstra

Pada gambar 5.6 adalah hasil pencarian jalur *multipath* antara h1 dan h2 menggunakan algoritme Dijkstra.

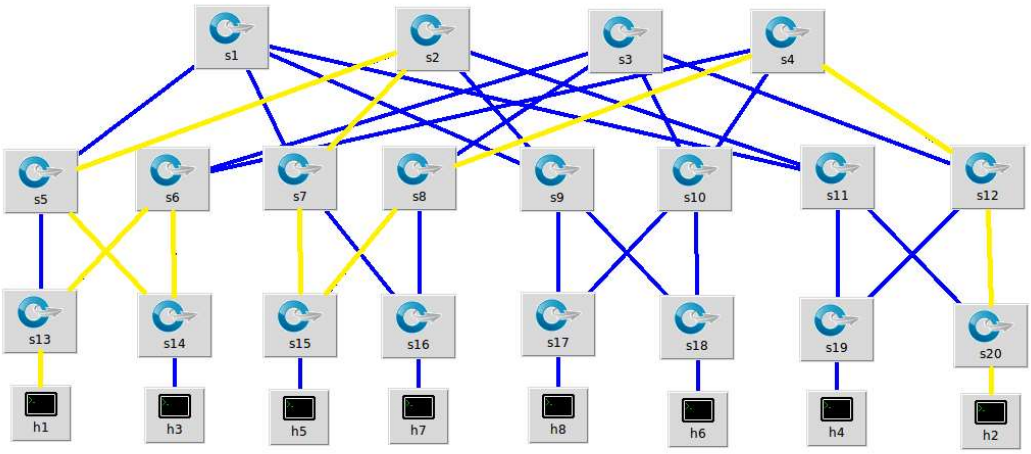
```
sdn@sdnvm:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s13
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=248.576s, table=0, n_packets=97, n_bytes=5820, priority=65535,dl_
dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=225.130s, table=0, n_packets=2, n_bytes=196, ip,nw_src=10.0.0.1,n
w_dst=10.0.0.2 actions=group:1114579192
 cookie=0x0, duration=225.130s, table=0, n_packets=1, n_bytes=42, priority=1,arp,arp_sp
a=10.0.0.1,arp_tpa=10.0.0.2 actions=group:1114579192
 cookie=0x0, duration=225.130s, table=0, n_packets=2, n_bytes=196, ip,nw_src=10.0.0.2,n
w_dst=10.0.0.1 actions=output:3
 cookie=0x0, duration=225.130s, table=0, n_packets=72, n_bytes=3024, priority=1,arp,arp
_spa=10.0.0.2,arp_tpa=10.0.0.1 actions=output:3
 cookie=0x0, duration=182.073s, table=0, n_packets=4, n_bytes=354, priority=1,ipv6 acti
ons=drop
 cookie=0x0, duration=248.648s, table=0, n_packets=11, n_bytes=508, priority=0 actions=
CONTROLLER:65535
```

Gambar 5.7 Hasil Instalasi *Flow*

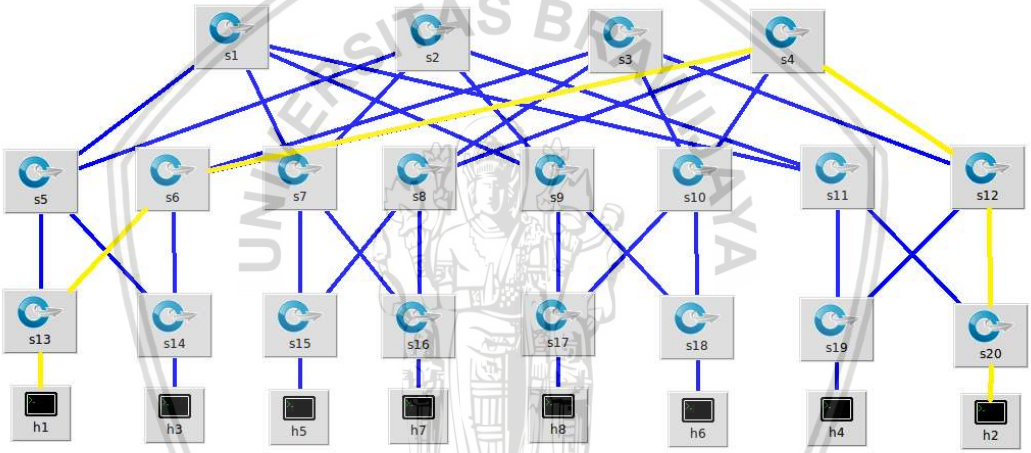
```
sdn@sdnvm:~$ sudo ovs-ofctl -O OpenFlow13 dump-groups s13
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=3945184109,type=select,bucket=weight:5,actions=output:1,bucket=weight:
5,actions=output:2
sdn@sdnvm:~$
```

Gambar 5.8 Hasil Instalasi *Group Table*

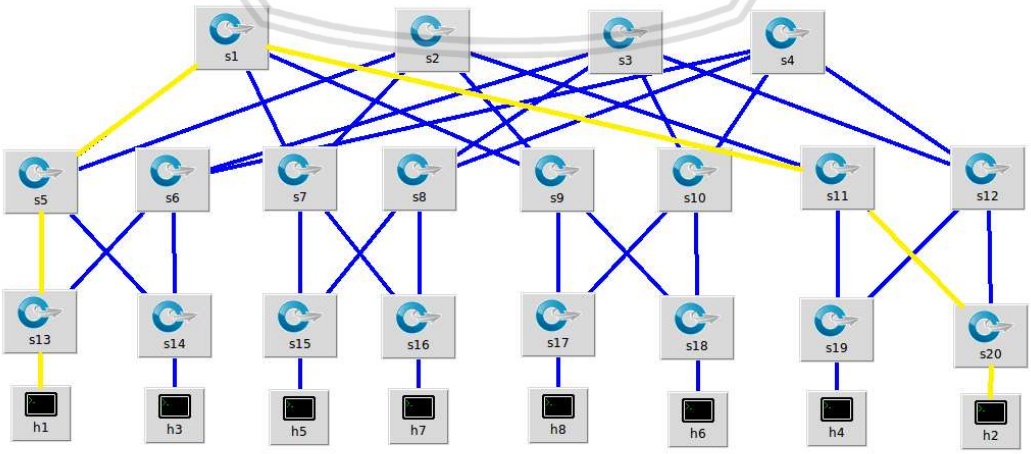
Pada gambar 5.8 adalah hasil instalasi *group table* pada *switch* 13 menggunakan tipe *select* yang memungkinkan untuk melakukan *load-balancing* dengan *action port* 1 dan 2.



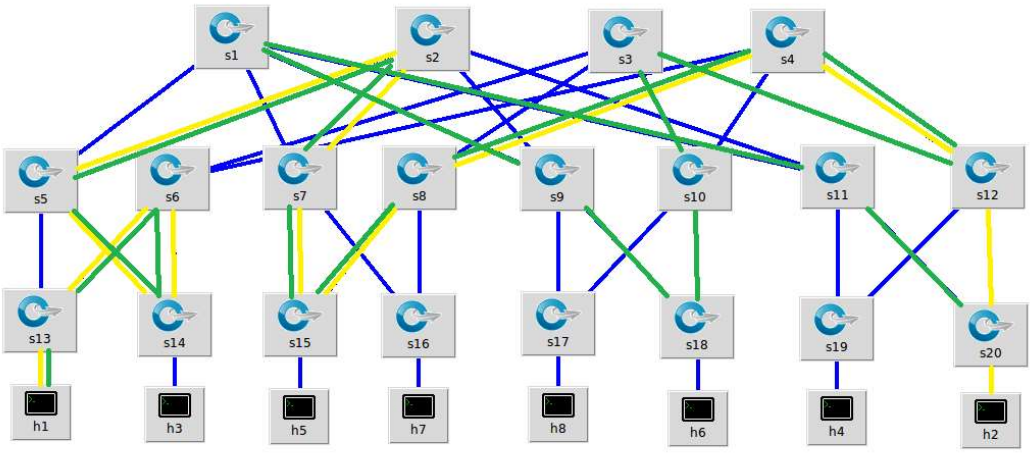
Gambar 5.9 Hasil Pencarian Jalur *Single-path* Menggunakan DFS



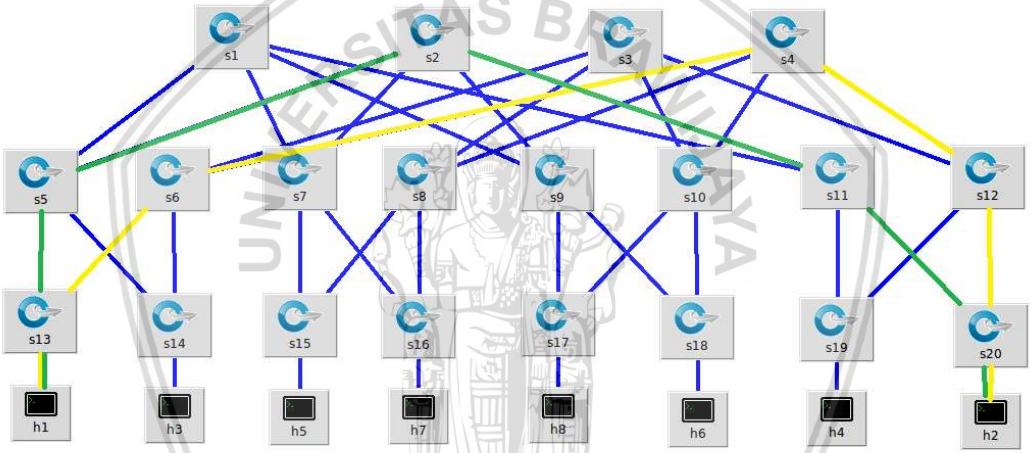
Gambar 5.10 Hasil Pencarian Jalur *Single-path* Menggunakan *Modified* DFS



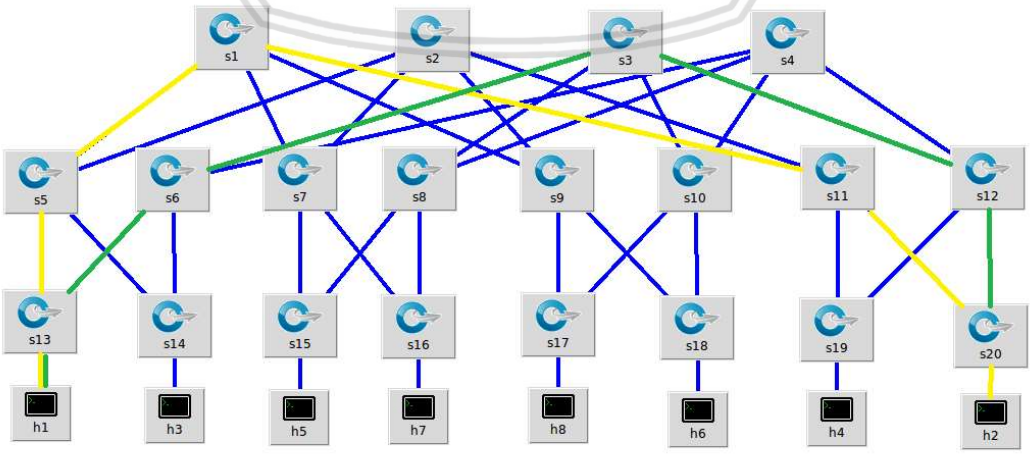
Gambar 5.11 Hasil Pencarian Jalur *Single-path* Menggunakan Dijkstra



Gambar 5.12 Hasil Pencarian Jalur *Multipath* Menggunakan DFS



Gambar 5.13 Hasil Pencarian Jalur *Multipath* Menggunakan *Modified* DFS



Gambar 5.14 Hasil Pencarian Jalur *Multipath* Menggunakan Dijkstra

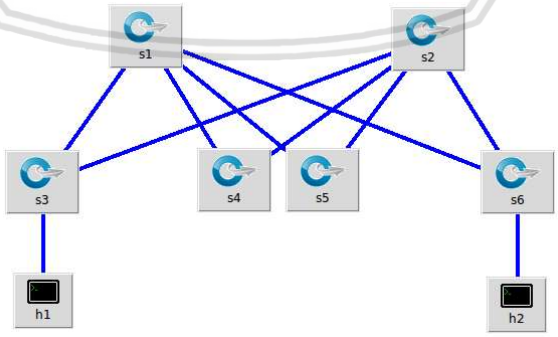
Tabel 5.1 Hasil Pengujian Pencarian Jalur

	DFS	*Modified* DFS	Dijkstra
Single-path	13, 6, 14, 5, 2, 7, 15, 8, 4, 12, 20	13, 6, 4, 12, 20	13, 5, 1, 11, 20
Multipath	[13, 6, 14, 5, 2, 7, 15, 8, 4, 12, 20], [13, 6, 14, 5, 2, 7, 15, 8, 4, 12, 3, 10, 18, 9, 1, 11, 20]	[13, 6, 4, 12, 20], [13, 5, 2, 11, 20]	[13, 5, 1, 11, 20], [13, 6, 3, 12, 20]

Berdasarkan hasil pengujian pencarian jalur yang telah dilakukan dari h1 menuju h2, dapat dilihat pada tabel 5.1 bahwa pencarian *single-path* menggunakan algoritme DFS yang tidak dimodifikasi menghasilkan jalur terjauh yaitu (13, 6, 14, 5, 2, 7, 15, 8, 4, 12, 20). Sedangkan dengan menggunakan algoritme DFS yang sudah dimodifikasi dapat menemukan jalur (13, 6, 4, 12, 20). Untuk pencarian *single-path* menggunakan Dijkstra didapatkan jalur (13, 5, 1, 11, 20). Pada pencarian *multipath* menggunakan DFS yang tidak dimodifikasi, dihasilkan jalur-jalur yang tidak independen seperti (13, 6, 14, 5, 2, 7, 15, 8, 4, 12, 20) dan (13, 6, 14, 5, 2, 7, 15, 8, 4, 12, 19, 11, 20) yang keduanya memiliki beberapa kesamaan pada jalurnya. Pada pencarian *multipath* menggunakan algoritme DFS yang dimodifikasi, dapat ditemukan jalur-jalur yang independen yaitu (13, 6, 4, 12, 20) dan (13, 5, 2, 11, 20). Sedangkan pada pencarian *multipath* menggunakan Dijktra ditemukan jalur (13, 5, 1, 11, 20) dan (13, 6, 3, 12, 20).

5.1.1 Pencarian Jalur Secara Manual

Berikut ini akan dijelaskan langkah-langkah pencarian jalur dengan algoritme DFS yang dimodifikasi dan Dijkstra.



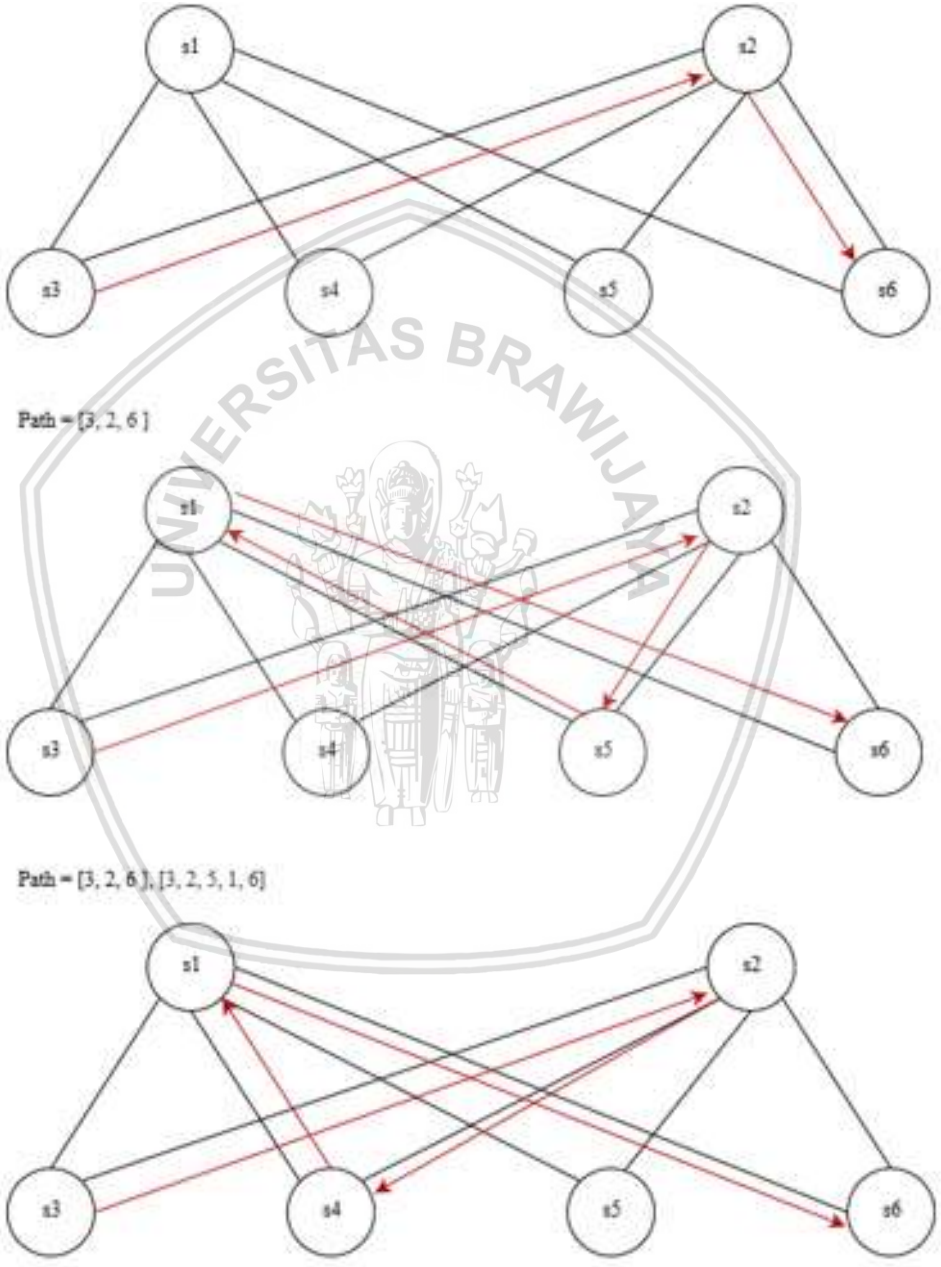
Gambar 5.15 Topologi *Fat Tree* 2 *Level*

Pada Gambar 5.14 adalah topologi *Fat Tree* 2 *Level*, topologi tersebut dapat direpresentasikan dalam sebuah struktur data *dictionary* untuk menyimpan informasi *adjacency* dari setiap *switch* sebagai berikut G = {1: {3: 1, 4: 2, 5: 3, 6: 4}, 2: {3: 1, 4: 2, 5: 3, 6: 4}, 3: {1: 1, 2: 2}, 4: {1: 1, 2: 2}, 5: {1: 1, 2: 2}, 6: {1: 1, 2: 2}}.

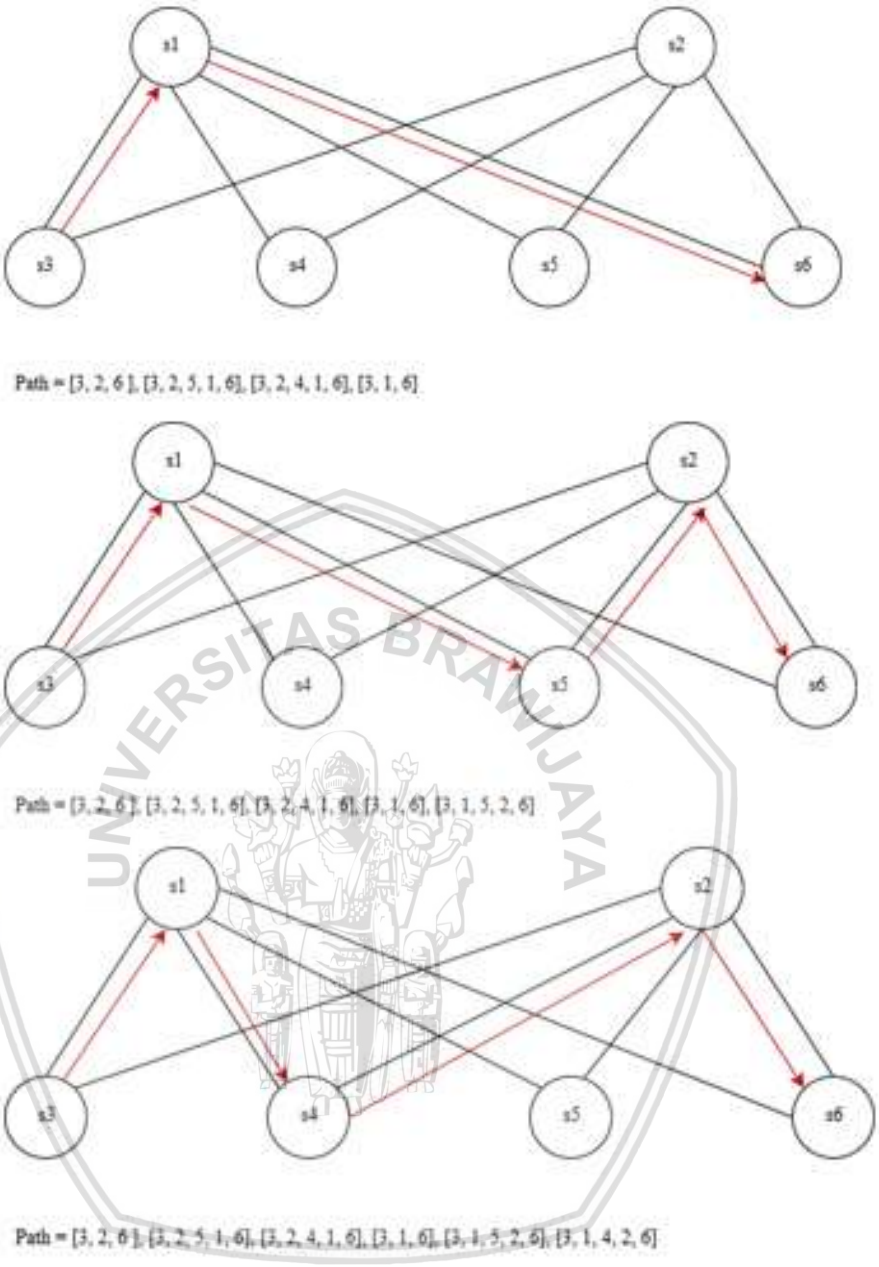
Pada struktur data G dapat diambil informasi seperti *switch 1* terhubung dengan *switch 3* melalui *port 1*. Berikut adalah proses pencarian jalur dari h1 ke h2 menggunakan DFS yang dimodifikasi dan Dijkstra.

a. *Modified*-DFS

- Menemukan semua jalur



Gambar 5.16 Proses Pencarian Jalur *Modified* DFS



Gambar 5.17 Proses Pencarian Jalur *Modified* DFS (Lanjutan)

- Mengurutkan jalur dari yang terpendek

  Path = [3, 2, 6], [3, 1, 6], [3, 2, 5, 1, 6], [3, 2, 4, 1, 6], [3, 1, 5, 2, 6], [3, 1, 4, 2, 6]

- Melakukan penyeleksian dari jalur yang ditemukan

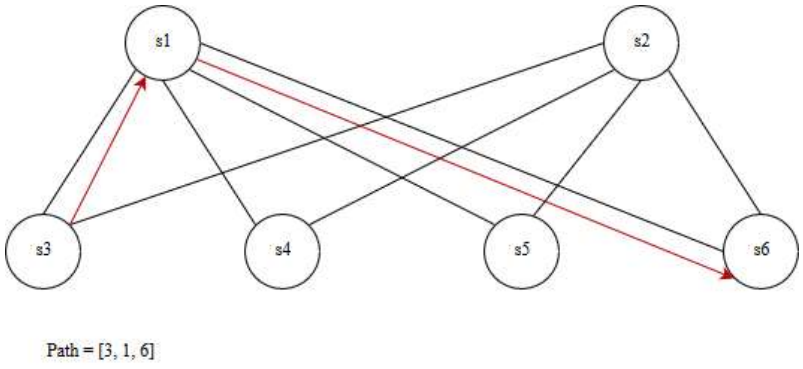
Tabel 5.2 Proses Seleksi Jalur *Modified* DFS

Iterasi	*Path*	*All Path*
0		[3, 2, 6] , [3, 1, 6], [3, 2, 5, 1, 6], [3, 2, 4, 1, 6], [3, 1, 5, 2, 6], [3, 1, 4, 2, 6]
1	[3, 2, 6]	[3, 1, 6], [**3**, **2**, 5, 1, 6], [3, 2, 4, 1, 6], [3, 1, 5, 2, 6], [3, 1, 4, 2, 6] Hapus jalur yang memiliki kesamaan [3, 1, 6], [3, 2, 4, 1, 6], [3, 1, 5, 2, 6], [3, 1, 4, 2, 6]
2	[3, 2, 6]	[3, 1, 6], [**3**, **2**, 4, 1, 6], [3, 1, 5, 2, 6], [3, 1, 4, 2, 6] Hapus jalur yang memiliki kesamaan [3, 1, 6], [3, 1, 5, 2, 6], [3, 1, 4, 2, 6]
3	[3, 2, 6]	[3, 1, 6], [3, 1, 5, **2**, **6**], [3, 1, 4, 2, 6] Hapus jalur yang memiliki kesamaan [3, 1, 6], [3, 1, 4, 2, 6]
4	[3, 2, 6]	[3, 1, 6], [3, 1, 4, **2**, **6**] Hapus jalur yang memiliki kesamaan [3, 1, 6]

- Dari proses penyeleksian jalur, didapatkan dua jalur yaitu [3, 2, 6] dan [3, 1, 6]

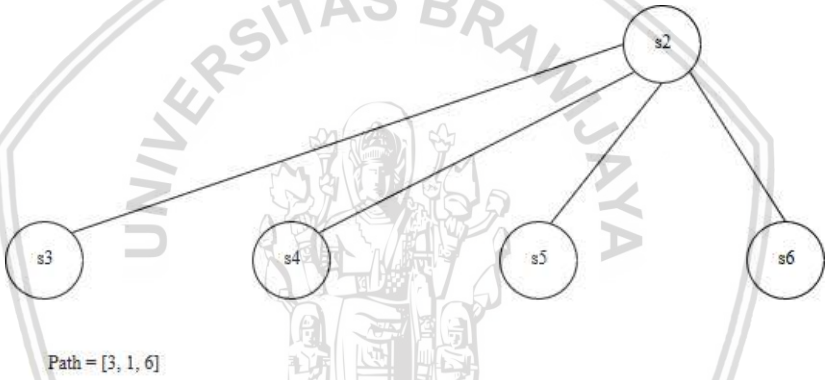
b. Dijkstra

- Iterasi 1



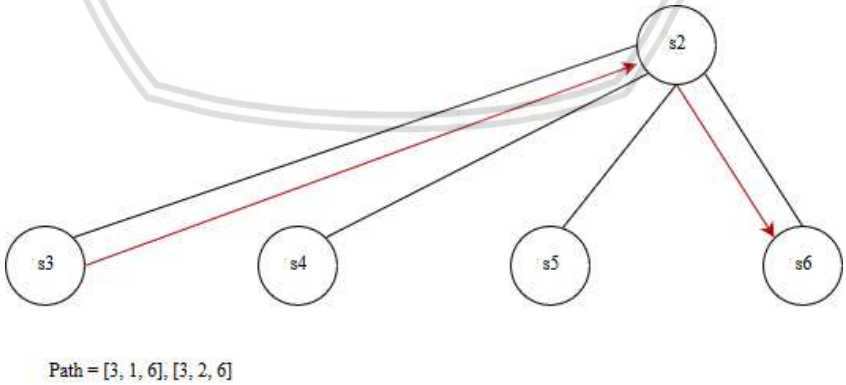
Gambar 5.18 Proses Pencarian Jalur Dijkstra Iterasi 1

- Hapus *node* dan *edge* yang telah ditemukan dari graf kecuali sumber dan tujuan.



Gambar 5.19 Proses Menghapus Jalur dari Graf Dijkstra Iterasi 1

- Iterasi 2



Gambar 5.20 Proses Pencarian Jalur Dijkstra Iterasi 2

- Hapus *node* dan *edge* yang telah ditemukan dari graf kecuali sumber dan tujuan

Gambar 5.21 Proses Menghapus Jalur dari Graf Dijkstra Iterasi 2

- Tidak ada jalur dari sumber ke tujuan, jalur-jalur yang ditemukan adalah [3,1,6] dan [3,2,6]

5.2 Pengujian *Response Time*

Response time didefinisikan sebagai waktu yang diperlukan *controller* dalam memproses paket pertama sampai berhasil mengistal *flow table* dan paket dapat dikirimkan. *Pada* pengujian *respone time,* dilakukan pengamatan pada *round trip time* dari pengiriman *ping* paket pertama saat program pertama kali dijalankan. Pengujian *response time* dilakukan sebanyak 10 kali pengujian.

Gambar 5.22 Contoh Hasil Ping dari h1 ke h2

Tabel 5.3 Hasil Pengujian *Response Time*

	DFS	*Modified*-DFS	Dijkstra
Rata-rata	612,7	632,6	325,6
Standar Deviasi	187,01	156,92	81,98

Pada tabel 5.3 merupakan hasil pengujian *response time*. Rata-rata *response time* algoritme DFS adalah 612,7 *ms.* Rata-rata *response time* algoritme DFS yang dimodifikasi adalah 632,6 *ms* dan Dijkstra 325,6 *ms.* Hasil *response time* menunjukkan bahwa algoritme DFS yang dimodifikasi memiliki waktu *response time* yang lebih lama dibandingkan algoritme *naïve* DFS. Sedangkan pada algoritme Dijkstra memiliki *response time* yang paling baik. Hal ini menunjukkan

bahwa proses *filtering* jalur pada DFS yang dimodifikasi membutuhkan waktu yang cukup lama.

5.3 Pengujian Jumlah Iterasi

Jumlah iterasi adalah banyak iterasi yang diperlukan algoritme pencarian jalur untuk menemukan n-jalur yang akan digunakan dalam *multipath routing.* Pengujian jumlah iterasi dilakukan untuk mengetahui kompleksitas dari suatu algoritme.

```
Running DFS Algorithm
Jumlah Iterasi:  12694
Path Execution Time :  0.0175220966339
[13, 6, 14, 5, 2, 7, 15, 8, 4, 12, 20] cost =  10.0
[13, 6, 14, 5, 2, 7, 15, 8, 4, 12, 19, 11, 20] cost =  12.0
Path installation finished in  0.0545799732208
```

Gambar 5.23 Jumlah Iterasi Algoritme DFS

```
Running DFS Algorithm
Jumlah Iterasi:  12860
Path Execution Time :  0.0221829414368
[20, 11, 2, 5, 13] cost =  4.0000006816
[20, 12, 4, 6, 13] cost =  4.00000139934
Path installation finished in  0.0900168418884
```

Gambar 5.24 Jumlah Iterasi Algoritme *Modified*-DFS

```
Running Dijkstra Algorithm
Iterasi:  400
Iterasi:  400
Path Execution Time :  0.000469923019409
[20, 11, 1, 5, 13] cost =  4
[20, 12, 3, 6, 13] cost =  4
Path installation finished in  0.00329613685608
```

Gambar 5.25 Jumlah Iterasi Algoritme Dijkstra

Tabel 5.4 Hasil Pengujian Jumlah Iterasi

Algoritme	DFS	*Modified*-DFS	Dijkstra
Jumlah Iterasi	12694	12860	800

Pada tabel 5.4 adalah hasil pengujian jumlah iterasi. Algoritme DFS memerlukan 12694 iterasi, *modified*-DFS sebanyak 12860 iterasi, dan Dijkstra sebanyak 800 iterasi. Algoritme DFS yang dimodifikasi memerlukan iterasi yang lebih banyak daripada algoritme *naïve* DFS. Hal ini dikarenakan penambahan iterasi untuk melakukan penyeleksian dari jalur-jalur yang ditemukan.

5.4 Pengujian *Execution Time*

Execution time adalah waktu yang dibutuhkan algoritme pencarian jalur untuk menemukan n-jalur yang akan digunakan dalam *multipath routing.* Pengujian *execution time* dilakukan bersamaan dengan pengujian *response time*.

```
Path installation finished in  0.00690913200378
Running Dijkstra Algorithm
Available paths from  20  to  13  :  [[20, 11, 1, 5, 13], [20, 12, 3, 6, 13]]
Path Execution Time :  0.000655174255371
[20, 11, 1, 5, 13] cost =  4.0
[20, 12, 3, 6, 13] cost =  4.0
Path installation finished in  0.00402998924255
Running Dijkstra Algorithm
Available paths from  13  to  20  :  [[13, 5, 1, 11, 20], [13, 6, 3, 12, 20]]
Path Execution Time :  0.00040602684021
[13, 5, 1, 11, 20] cost =  4.0
[13, 6, 3, 12, 20] cost =  4.0
Path installation finished in  0.0954999923706
```

Gambar 5.26 Contoh Hasil Waktu Eksekusi Pencarian Jalur dengan DFS

Tabel 5.5 Hasil Pengujian *Execution Time*

Pengujian	DFS (*ms*)	*Modified* DFS (*ms*)	*Modified* Dijkstra (*ms*)
1	0,0187	0,0312	0,0004
2	0,0165	0,0217	0,0005
3	0,0170	0,0245	0,0004
4	0,0183	0,0211	0,0004
5	0,0201	0,0225	0,0004
6	0,0172	0,0291	0,0007
7	0,0186	0,0250	0,0004
8	0,0182	0,0328	0,0004
9	0,0176	0,0221	0,0004
10	0,0230	0,0284	0,0004
Rata-rata	0,0185	0,0258	0,0005

Pada tabel 5.5 merupakan hasil pengujian *execution time*. Rata-rata algoritme DFS adalah 0,0185 *ms, modified* DFS adalah 0,0258 *ms*, dan *modified* Dijkstra 0,0005 *ms.*

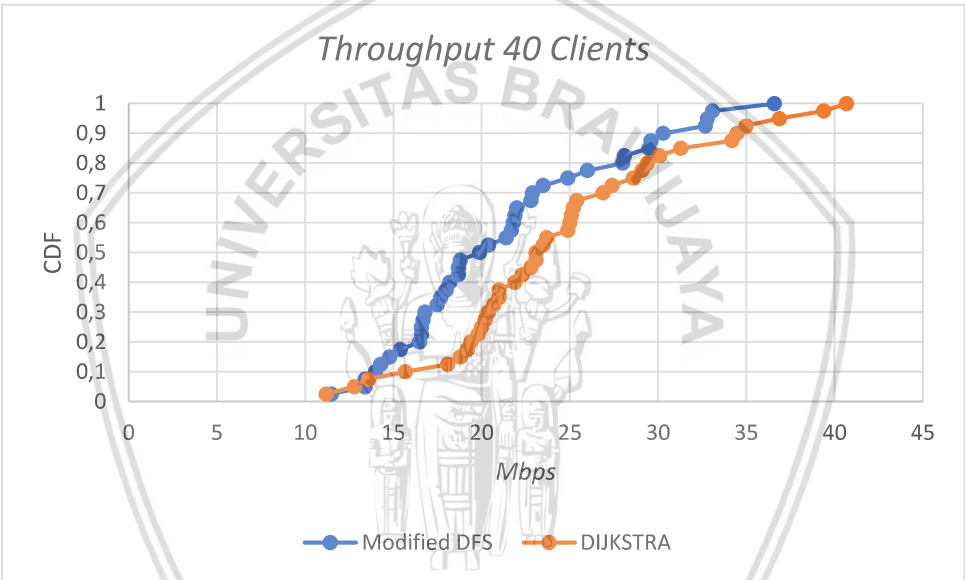
5.5 Pengujian *Throughput*

Pada pengujian *througput* dilakukan dengan menggunakan *iperf* antara h1 h2, h3 h4, h5 h6, dan h7 h8. Pada pengujian ini, h1, h3, h5, dan h7 bertindak sebagai

server sedangkan h2, h4, h6, h8 bertindak sebagai *client. Host* yang bertindak sebagai *client* mengirimkan *traffic* TCP selama 90 detik dengan menggunakan jumlah koneksi *client* sebanyak 10 dan *bitrate* masing-masing *client* adalah 100 *Mbps*, sehingga total *client* yang digunakan adalah 40 *client.*

```
root@sdnvm:~# iperf3 -c 10.0.0.1 -P 10 -b 100M
Connecting to host 10.0.0.1, port 5201
[  6] local 10.0.0.2 port 44968 connected to 10.0.0.1 port 5201
[240] local 10.0.0.2 port 44970 connected to 10.0.0.1 port 5201
[242] local 10.0.0.2 port 44972 connected to 10.0.0.1 port 5201
[244] local 10.0.0.2 port 44974 connected to 10.0.0.1 port 5201
[246] local 10.0.0.2 port 44978 connected to 10.0.0.1 port 5201
[248] local 10.0.0.2 port 44980 connected to 10.0.0.1 port 5201
[250] local 10.0.0.2 port 44982 connected to 10.0.0.1 port 5201
[252] local 10.0.0.2 port 44984 connected to 10.0.0.1 port 5201
[254] local 10.0.0.2 port 44986 connected to 10.0.0.1 port 5201
[256] local 10.0.0.2 port 44988 connected to 10.0.0.1 port 5201
```

Gambar 5.27 Contoh Pengujian *Throughput* dengan 10 *Client*

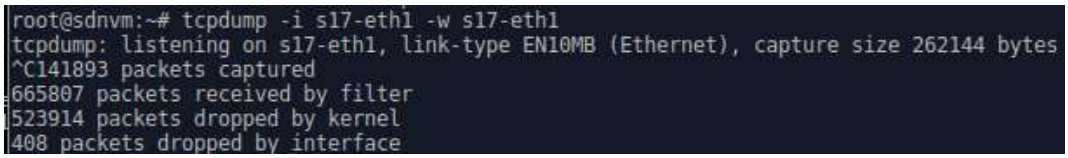


Gambar 5.28 Hasil Pengujian *Throughput* dengan 40 *Client*

Pada gambar 5.28 merupakan data dari pengujian *throughput* dengan 40 *client* yang direpresentasikan dalam grafik CDF. Dari grafik tersebut dapat diketahui sebanyak 50% dari *client* mendapat *throughput* 20 *Mbps* atau kurang pada algoritme *modified*-DFS. Sedangkan pada Dijkstra, sebanyak 50% dari *client* mendapat *throughput* 23 *Mbps* atau kurang. Pada pengujian ini Dijkstra memiliki *throughput* yang lebih baik.

5.6 Pengujian *Multipath*

Pengujian *multipath* dilakukan untuk mengetahui apakah sistem dapat membagi *flow* melaui beberapa jalur yang tersedia. Pengujian dilakukan bersamaan dengan pengujian *throughput* dengan menangkap *packet* pada *switch* dan *port* yang melakukan *multipath* menggunakan tcpdump.



Gambar 5.29 Contoh Penggunaan tcpdump

Tabel 5.6 Hasil Pengujian *Multipath*

Algoritme	S20		S19		S18		S17	
	Port 1	Port 2	Port 1	Port 2	Port 1	Port 2	Port 1	Port 2
Modified-DFS	5	5	7	3	5	5	7	3
Dijkstra	7	3	5	5	5	5	5	5

Berdasarkan hasil pengujian *multipath* yang dilakukan, pada algoritme *modified*-DFS di *switch* 19 dan 17 *flow* tidak didistribusikan seimbang yaitu terdapat 7 *flow* pada salah satu jalur dan 3 *flow* di jalur lainnya. Sedangkan pada algoritme Dijkstra di *switch* 20 *flow* tidak terdistribusi dengan baik yaitu sebanyak 7 *flow* pada *port* 1 dan 3 *flow* pada *port* 2.

BAB 6 PENUTUP

6.1 Kesimpulan

Berdasarkan hasil pengujian dan analisis, didapatkan kesimpulan dari penelitian ini sebagai berikut.

1. Pencarian n-jalur *multipath* pada jaringan OpenFlow dapat menggunakan algoritme DFS dengan melakukan modifikasi sehingga ditemukan seluruh jalur dari suatu sumber ke tujuan. Kemudian jalur-jalur tersebut diurutkan berdasarkan *hop* terpendek dan dilakukan seleksi pada tiap jalur untuk menemukan jalur-jalur yang independen atau *disjoint*.
2. Dalam melakukan *load-balancing* pada jalur yang telah ditemukan dilakukan perhitungan *weigth* yang dijadikan acuan dalam mendistribusikan *traffic* menggunakan mekanisme *load-balancing* yang sediakan oleh OpenFlow menggunakan *group table* tipe *select*.
3. Berdasarkan hasil pengujian algoritme *modified* Dijkstra memiliki kinerja yang paling baik. Sedangkan algoritme *modified* DFS membutuhkan waktu eksekusi lebih lama dari *naïve* DFS namun dapat ditemukan jalur-jalur yang tidak memiliki kesamaan.

6.2 Saran

Berikut adalah saran-saran yang dapat digunakan untuk melakukan pengembangan selanjutnya.

1. Program *controller* perlu dilakukan pengembangan agar dapat menentukan *weigth* dan jalur secara dinamis.
2. Pada penelitian ini mekanisme *load-balancing* masih menggunakan perhitungan *hash* yang disediakan oleh OpenFlow 1.3 dengan menggunakan *group table* tipe *select*. Mekanisme *load-balancing* perlu dilakukan pengembangan lagi.
3. Program *controller* perlu dikembangkan untuk mengatasi kegagalan *link*.

DAFTAR PUSTAKA

Chiang, Y.-R.et al., 2017. *A Multipath Transmission Scheme for the Improvement of Throughput over SDN.* Proceedings of the 2017 IEEE International Conference on Applied System Innovation, IEEE.

Cisco, 2005. *OSPF Design Guide.* [online] Tersedia di: <https://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/7039-1.html> [Diakses 4 November 2017]

Hakiri, A. et al., 2014. Software-Defined Networking: Challenges and research opportunities for Future Internet. *Computer Networks,* Volume Volume 75, pp. 453-471.

Heap, D., 2002. *Depth First Search (DFS)*. [online] Tersedia di: <http://www.cs.toronto.edu/~heap/270F02/node36.html> [Diakses 30 Oktober 2017]

InMon, 2017. sFlow-RT. [online] Tersedia di: <http://www.sflow-rt.com/> [Diakses 31 Oktober 2017]

Jiang, J.-R., Huang, H.-W. & Liao, J.-H., 2014. *Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking.* Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific, IEEE.

Joe, E., Pan, D., Liu, J. & Butler, L., 2014. *A Simulation and Emulation Study of SDN Based Multipath Routing for Fat-Tree Data Center Networks.* Proceedings of the 2014 Winter Simulation Conference, IEEE.

Kim, H. & Feamster, N., 2013. Improving Network Management with Software Defined Networking. *IEEE Communications Magazine,* 51(2), pp. 114 - 119.

Kreutz, et al., 2015. *Software-Defined Networking: A Comprehensive Survey.* s.l., IEEE.

Lei, Y.-C., Wang, K. & Hsu, Y.-H., 2015. *Multipath Routing in SDN-based Data Center Networks.* European Conference on Networks and Communications (EuCNC), IEEE.

Linux Fundation, 2016, *OpenvSwitch*. [online] Tersedia di: <http://openvswitch.org/> [Diakses 31 Oktober 2017]

Maulana, W., 2017. Multipath Routing dengan Load-Balancing Pada OpenFlow Software-Defined Network. *Repositori Jurnal Mahasiswa PTIIK UB,* Volume 9, p. 15.

Mininet Team, 2017. [online] Tersedia di: <http://mininet.org/overview/> [Diakses 31 Oktober 2017]

Mustafa, M. E. & Ibrahim, A. M. A., 2015. Load Balancing Algorithms Round (RR), Least-Connection and Least Loaded Efficiency. *International Journal of Computer and Information Technology*

Nunes, B. A. A. et al., 2014. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys & Tutorials,* 16(3), pp. 1617 - 1634.

Open Network Foundation (ONF) (ONF white paper). [online] Tersedia di: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf> [Diakses 30 Oktober 2017]

Open Network Foundation (ONF) (ONF white paper). [online] Tersedia di: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf> [Diakses 30 Oktober 2017]

Open Networking Foundation, 2012. OpenFlow *Switch Specification* Version 1.3.0. [pdf] Tersedia di: <https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf> [Diakses 30 Oktober 2017]

Ramdhani, M. F., Hertiana, S. N. & Dirgantara, B., 2016. *Multipath Routing with Load Balancing and Admission Control in Software Networking (SDN).* Fourth International Conference on Information and Communication Technologies (ICoICT), IEEE.

Ryu SDN Framework Community, 2017. [online] Tersedia di: <https://osrg.github.io/ryu/> [Diakses 31 Oktober 2017]

Wang, T., Su, Z., Xia, Y. & Hamdi, M., 2014. Rethinking the Data Center Networking: Architecture, Network Protocols, and Resource Sharing. *IEEE Access,* Volume 2, pp. 1481-1496.